Rešene naloge iz programiranja v Pythonu

Janez Demšar
Jure Žbontar
Martin Možina
Tomaž Hočevar

Uvod

Znano je, da dandanašnji študenti ne berejo uvodov. Nič ne de, dandanašnji profesorji jih še vedno radi pišejo. S tem je pač enako kot s predavanji.

Zbirka je nastala iz

a) frustracije avtorjev nad neobstojem podobnih zbirk; kar je takšnih nalog, so pogosto preveč matematično orientirane (komu mar Stirlingova števila druge vrste!),

b) lastne zaloge nalog, ki jih uporabljamo za frustriranje študentov na vajah in za metanje na izpitih.

Matematičnim nalogam smo se poskušali izogniti, a povsem ne gre. Čisti začetnik zna premalo, da bi obračal kaj drugega kot številke. Poleg tega so naloge, kot so iskanje praštevil in popolnih števil, preveč preproste in poučne hkrati, da bi se jim mogli odreči.

Običajne zbirke nalog – če smemo uporabiti terminologijo iz reklam za pralne praške – so navadno splošne, primerne za poljuben jezik. Še več, pogosto so primerne predvsem za prevajane jezike, kot sta C in Java; v Pythonu lahko začetnik reši veliko zanimivejše naloge, zato je škoda, da bi pozdravljal svet in risal smrekice iz zvezdic. Vsaj prvih, splošnejših nalog, se lahko brez skrbi lotite tudi v poljubnem drugem jeziku, kasnejše pa boste z lahkoto reševali le še v jezikih, v katere so tesno vdelani seznami, slovarji in funkcije za delo z nizi.

V točki b) gre seveda za popolno pretiravanje. Naši študenti hodijo z izpitov nasmejani. Na vaje pa sploh ne.

Ali pač. Po pravici povejmo: programiranje zahteva dril. Kdor upa, da je s tem tako kot z nasledniki Karla Velikega, ki smo se jih lahko naučili dan pred kontrolko iz Zgodovine (in pozabili na pivu po njej), živi v nevarni in potencialno usodni zmoti. Učenje programiranja je potrebno jemati kot trening. Naloge rešujte po potrebi tudi večkrat, toliko časa, dokler ne postanejo rutinske. Po naši oceni so lahke, potrebuje pa jih prav tisti, ki se z našo oceno ne strinja.

Rešitve so namerno ločene od nalog. Druga huda zmota, poleg te, da je učenje programiranja podobno učenju imen Šarlemajnovih potomcev, je, da je razumeti rešitev enako dobro kot znati rešiti. Ni. Rešitev je najboljše pogledati šele, ko je naloga (vsaj približno uspešno) rešena. Takrat pa jo le dobro poglejte, saj njen namen ni pokazati, kako se reši nalogo, temveč, kako se jo reši lepo (in tudi, kako se je ne sme reševati). Ko ste nalogo rešili in prebrali rešitev, poskusite še sami napisati enak program. Ne pretipkati, na novo napisati.

Do znanja programiranja ni bližnjic.

Še en motiv za zbirko nalog (kot alternativo učbeniku) je v tem, da učbeniki učijo. Programiranja pa se ne uči, temveč trenira. Pomemben del treninga je analiza, kaj sem naredil

dobro in česa ne. Zbirke nalog, ki ne vsebujejo rešitev, še bolj pa tiste, v katerih so rešitve brez komentarjev, zanemarjajo pomemben del svojega poslanstva. Ta zbirka ga ne. Dojemajte jo, tako kot mi, manj kot zbirko nalog in bolj kot zbirko njihovih rešitev.

Knjigo berite odprto. Če naloga zahteva, da poiščete vse pare, se vprašajte, ali znate najti tudi trojke. Če zahteva, da poiščete največjega, razmislite, ali znate izpisati tudi največjih pet.

Naloge so urejene po nekakšnih sklopih, rešitve pa ne. Eno in isto nalogo bo bolj izkušen programer rešil preprosteje kot začetnik. Pokazali smo obe rešitvi (ali pa vseh osem); boste že sami vedeli, katerim od njih ste že dorasli. Že ob prvih nalogah smo včasih napisali tudi rešitev, ki je ne razume niti študent po celem letu programiranja. To naj bo namig, da se splača, ko se prebijete do konca, ponovno lotiti preprostih nalog: morda jih boste znali rešiti boljše, kot ste jih zmogli prvič. Tudi po težavnosti se jih nismo trudili urejati: nekaterim je lažje eno, drugim drugo. Sploh pa se da mnoge naloge reševati na vedno nove, boljše načine, torej ni nič narobe, če na kako nalogo, ki bi jo lahko dali med lažje, naletimo kasneje; bomo pač našli boljšo rešitev, ko bi jo pred dvajsetimi nalogami.

V rešitvah smo poskušali biti elegantni in poučni. Veliko problemov bi izkušen programer odpravil s priročnim trikom ali uporabo eksotične vdelane funkcije. To ni bil naš namen in kjer ni preveč klicalo po tem, se s takimi rešitvami nismo šopirili. Po drugi strani pa jih tudi nismo skrivali, kjer so koristne.

Nekatere rešitve se sklicujejo na rešitve predhodnih nalog, spet druge sklerotično ponavljajo isto reč, kot jo je razlagala že rešitev dveh nalogi višje. To je zato, ker naloge lahko rešujete po vrsti ali pa tudi ne. Kakor se vam zahoče. Rešitve so pisane skladno s tem: kakor se nam je zahotelo.

V zbirki ni nalog, povezanih s predmetno usmerjenim programiranjem. Za ta format niso primerne. Prehitro se sprevržejo v manjše projekte, kot je na primer razred za delo z ulomki. Ti sodijo v učbenike. Temu se da izogniti tako, da pripravimo razred, na primer razred za risanje z želvo, in v nalogah zahtevamo takšne in drugačne dodatke. To dobro deluje v učilnici, v knjigi pa ne. Izognili smo se tudi nekaterim drugim temam, ki so lahko del osnovnih tečajev programiranja, denimo uporabniškim vmesnikom in risanju. Oboje je zabavno, je pa specifično in bolj kot trening programiranja predstavlja spoznavanje knjižnic.

Pač pa smo v primerjavi s tečaji za začetnike posvetili nenavadno veliko nalog preobračanju besedil, iskanju po datotekah in podobnemu. Python je kot skriptni jezik za to posebej priročen in prav se nam ga zdi predstaviti tudi s te strani, da se ve, da takrat, ko je treba premetati kako besedilo, v resnici ni izbire med Pythonom, Cjem in Javo (je pa, seveda, med Pythonom, Rubyjem in podobnimi skriptnimi jeziki). Obenem pa nismo zarinili pregloboko v regularne izraze. Zanje bi bilo mogoče narediti ločeno vadnico, vendar ne bi bila posebej privlačna. V tej zbirki jih zato le bežno omenjamo.

Uvod za učitelje

Zbirka je z leti postala že kar obsežna in zna biti, da jo bo začel uporabljati tudi kak učitelj. Dobrodošel. Tule je nekaj *insiderskih* povedi zanj.

Če je dotični učitelj vajen kakih starejših jezikov, od pascala do C-ja: Python je drugačen, zato ga tudi učimo drugače. Po sedmih letih lastnih izkušenj – in izkušnjah iz boljših knjig – se splača hitro uvesti spremenljivko in nato pogoje, iz tega pa naravno sledi zanka `while`. `If` je »če«, `while` pa »dokler«. »If« se zgodi (kvečjemu) enkrat, »while« pa tolikokrat, kolikorkrat pogoj ostaja izpolnjen.

Zanka `for` v Pythonu ni takšna kot v C-ju ali pascalu, kjer gre bolj za `while` z malo drugačno sintakso. Zato tudi njegova obravnava ne sodi v isti koš. Pythonov `for` je podoben temu, čemur se v nekaterih jezikih pravi »for each«. Njen naravni habitat so torej nizi in seznami (ter druge reči, prek katerih je mogoče *iterirati*). Zbirka je zastavljena tako, kot zadnja leta zastavljamo tudi predavanja: zanko `for` najprej treniramo na seznamih in nizih, celo `enumerate` in `zip` se naučimo že kar tu (pokazalo se je, da tudi začetniki z njima nimajo težav!). Vse zato, da jih odvadimo brez potrebe uporabljati `for i in range(len(s))` in potem `e = s[i]` namesto preprostega, preglednega, `for e in s`. Zanke prek številskih intervalov in indeksiranje so namerno na vrsti šele, ko je pravilna raba `for` že dobro utrjena. Sledijo vaje, v katerih sezname spreminjamo.

Naloge so torej razdeljene tematsko, ne po težavnosti. Nekatere naloge iz zgodnejših sklopov so lahko težje od teh iz kasnejših. Sploh pa je vsako mogoče rešiti na težje ali lažje načine.

Posebnih nalog iz pisanja funkcij v zbirki ni. Tiho jih začnemo uporabljati enkrat pred zankami `for`. Prej po njih ni potrebe.

Sledijo naloge iz slovarjev in množic, nato pa splošnejše naloge, pri katerih je potrebno večinoa uporabljati le sezname in nize. V resnici pa gre za splošno programersko telovadbo. Šele ko je ta za nami, se lotimo najtežje stvari na svetu, rekurzije.

Zadnje poglavje, premetavanje nizov in besedil, je precej specifično za Python. Če želimo učiti programiranje v splošnem, ga lahko preskočimo. Po drugi strani pa se prav ob njem naučimo veliko praktično uporabnih stvari.

Knjiga spremlja predmet Programiranje 1, kot se predava na Visokošolskem strokovnem študiju na Fakulteti za računalništvo in informatiko ter na Pedagoški fakulteti. Zapiski predavanj so objavljeni na strani https://ucilnica.fri.uni-lj.si/p1.

Seznam nalog

Čisti začetek

1. Pretvarjanje iz Fahrenheitov v Celzije
2. Pitagorov izrek
3. Topologija
4. Ploščina trikotnika

Pogoji in zanke

5. Vaja iz poštevanke
6. Vsi po pet
7. Konkurenca
8. Top-shop
9. Državna agencija za varstvo potrošnikov
10. Collatzova domneva
11. Benjaminovi kovanci
12. Tekmovanje iz poštevanke
13. Števke
14. Obrnjena števila
15. Kalkulator

Zanke prek seznamov in nizov

16. Vsota elementov seznama
17. Ajavost nizov
18. Največji element
19. Največji absolutist
20. Najmanjši pozitivist
21. Najdaljša beseda
22. Poprečje
23. Poprečje brez skrajnežev
24. Bomboni
25. Vsaj eno liho
26. Sama liha
27. Blagajna
28. Preobremenjeni čolni

Zanke prek številskih intervalov

29. Delitelji
30. Praštevilo
31. Vsota deliteljev
32. Popolno število
33. Vsa popolna števila
34. Prijateljska števila
35. Vsebuje 7
36. Poštevanka števila 7
37. Fibonaccijevo zaporedje
38. Evklidov algoritem

Zanke prek več reči hkrati

39. Indeks telesne teže
40. Seštete trojke
41. Skalarni produkt
42. Ujemanja črk
43. Vzorec besede
44. Prva beseda
45. Paralelni skoki
46. Mesto največjega elementa
47. Olimpijske medalje
48. Vstavi teže
49. Primerjanje seznamov

Indeksiranje, seznami, nizi

50. Spol iz EMŠO
51. Pravilnost EMŠO
52. Starost iz EMŠO
53. Domine
54. Dan v letu
55. Nepadajoči seznam
56. Mesta črke
57. Multiplikativni range
58. Sumljive besede
59. Kockarji
60. Križanka
61. Sosedi
62. Glajenje
63. An ban pet podgan
64. Največji skupni delitelj seznama
65. Oškodovani otroci
66. Lomljenje čokolade
67. Razcep na praštevila
68. Pari števil
69. Ploščina poligona
70. Sodost čebel

71. Sodi – lihi
72. Najprej lihi
73. Sodo in liho in sodo in liho
74. Plus - minus - plus
75. Pecivo
76. Nogavice brez para
77. Presedanje

Slovarji in množice

78. Pokaži črke
79. Podobna beseda
80. Število znakov
81. Najpogostejša beseda in črka
82. Samo enkrat
83. Popularni rojstni dnevi
84. Osrednja obveščevalna služba
85. Menjave
86. Anagrami
87. Bomboniera
88. Prafaktorji in delitelji
89. Hamilkon
90. Družinsko drevo
91. Naključno generirano besedilo
92. Grde besede
93. Kronogrami
94. Posebnež
95. Sumljive besede
96. Transakcije
97. Natakar
98. Tečaji
99. Lego
100. Slepa polja
101. Inventar
102. Nedostopna polja
103. Opravljivke
104. Vir in ponor
105. Viri
106. Sociogram
107. Izplačilo
108. Dopisovalci
109. Zaporniki
110. Ograje
111. Trgovanje
112. Ne na lihih
113. Sopomenke
114. Stavka z istim pomenom
115. Združi - razmeči
116. Podajanje daril
117. Požrešneži
118. Ne maram
119. Najmanjši unikat
120. Bingo
121. Trki besed

Rekurzija

122. Preštej vnuke
123. Poišči rojaka
124. Poišči potomca
125. Preštej rodbino
126. Preštej potomce
127. Najdaljše ime v rodbini
128. Globina rodbine
129. Kolikokrat ime
130. Koliko žensk
131. Naštej rodbino
132. Naštej potomce
133. Največ otrok
134. Največ vnukov
135. Največ sester
136. Najplodovitejši
137. Brez potomca
138. Vsi brez potomca
139. Kako daleč
140. Pot do
141. Zaporedja soimenjakov
142. Fakulteta
143. Fibonacijeva števila
144. Vsota seznama
145. Iskanje elementa
146. Enaka seznama
147. Palindrom
148. Preverjanje Fibonacija
149. Naraščajoči seznam
150. Vsota gnezdenega seznama
151. Obrni
152. Zrcalo

153. Sodo – lihi – rekurzivno
154. Kam?
155. Razdalja do cilja
156. Pot do cilja
157. Naprej nazaj
158. Aritmetično zaporedje
159. Binarno
160. Decimalno
161. Zadnje liho
162. Indeksi
163. Brez jecljanja
164. Rekurzivni Collatz
165. Sprazni
166. Rekurzivni štumfi, zokni, kalcete, kucjte

Splošne vaje iz programiranja

167. Stopnice
168. Drugi največji element
169. Collatz 2
170. Delnice
171. Spremembe smeri
172. Sekajoči se krogi
173. Največ n-krat
174. Brez n-tih
175. Vse črke
176. Skritopis
177. Igra z besedami
178. Srečanje čebel
179. Odvečni presledki
180. Kričiš!
181. Napadalne kraljice
182. Obljudeni stolpci
183. Banka
184. Srečni gostje
185. Gostoljubni gostitelji
186. Po starosti
187. Ujeme
188. Najboljše prileganje podniza
189. Deljenje nizov
190. Bralca bratca
191. Turnir Evenzero
192. Najdaljše nepadajoče zaporedje
193. Seznam vsot seznamov
194. Veliko, a ne več kot
195. Nepadajoči podseznami
196. Sodi vs. lihi
197. Gnezdeni oklepaji
198. Črkovni oklepaji
199. Brez oklepajev
200. Vsota kvadratov palindromnih števil
201. Skupinjenje
202. Trdnjava
203. Vsote peterk
204. Legalni konj
205. Skoki
206. Bobri plešejo
207. Bobri vsi domov
208. Najbogatejši cvet
209. Pravilna pot
210. Dobiček na poti
211. Enkratna pot
212. Neobrani cvetovi
213. Žabji skoki
214. Lov na muhe
215. Žabja restavracija
216. Poštar iz Hamiltona
217. Graf hamiltonskega poštarja
218. Pretakanje
219. Slovar anagramov
220. Človek ne jezi se
221. Največ dvakrat
222. Seštej zaporedne
223. Intervali
224. Dolžine ladij

Premetavanje nizov in besedil

225. Sekunde
226. Naslednji avtobus
227. Eboran
228. Cenzura
229. Črkovalnik
230. Hosts
231. Uporabniške skupine
232. Skrito sporočilo
233. Iskanje URLjev
234. Deli URLja

235. Trgovinski računi
236. Izračun računa
237. Numerologija
238. Grep
239. Preimenovanje datotek
240. Vse datoteke s končnico .py
241. Uredi CSV
242. Zaporedni samoglasniki in soglasniki
243. Migracije
244. Selitve
245. Navodila
246. Poet tvoj nov Slovencem venec vije

Naloge

Čisti začetek

1. Pretvarjanje iz Fahrenheitov v Celzije

Napiši program, ki mu uporabnik vpiše temperaturo v Fahrenheitovih stopinjah, program pa jo izpiše v Celzijevih. Med temperaturama pretvarjamo po formuli C = 5/9 (F – 32). Za potrebe Američanov napiši še program, ki računa v nasprotno smer.

Če se k tej nalogi vračaš, da jo uporabiš za vajo iz pisanja funkcij, pa napiši funkciji `celsius(f)` in `fahrenheit(c)`. Prva prejme temperaturo v Fahrenheitih in vrne temperaturo v Celzijevih stopinjah, druga pa ravno obratno.

2. Pitagorov izrek

Napiši program, ki uporabnika vpraša po dolžinah katet pravokotnega trikotnika in izpiše dolžino hipotenuze. (Da ponagajamo pokojnemu prof. Križaniču, ki je sovražil dlakocepce, dodajmo "dolžino hipotenuze taistega trikotnika".)

Če nalogo uporabljaš za vajo iz pisanja funkcij, napiši funkcijo `pitagora(a, b)`, ki prejme dolžini katet in vrne dolžino hipotenuze.

3. Topologija

Napiši program za izračun dolžine strela s topom (ki brez trenja izstreljuje točkaste krogle v brezzračnem prostoru, a pustimo trivio). Program od uporabnika ali uporabnice zahteva, da vnese hitrost izstrelka (to je, omenjene točkaste krogle) in kot, pod katerim je izstreljen. Program naj izračuna in izpiše, kako daleč bo letela krogla.

Pomoč za fizično nebogljene: $s = v^2 \sin(2\phi)/g$, kjer je s razdalja, v hitrost izstrelka, ϕ je kot, g pa osma črka slovenske abecede.

Preveri tole: krogla leti najdalj, če jo izstrelimo pod kotom 45 stopinj. Poskusi, kako daleč gre pod kotom 45 in kako daleč pod 46 stopinj -- po 45 mora leteti dlje. Preveri tudi kot 50 stopinj: če pod tem kotom leti nazaj (razdalja je negativna), si ga gotovo nekje polomil.

Če boš napisal rešitev v obliki funkcije, naj bo to funkcija `top(v, fi)`.

4. Ploščina trikotnika

Ploščino trikotnika s stranicami a, b in c lahko izračunamo po Heronovem obrazcu: $p = \sqrt{s(s-a)(s-b)(s-c)}$, kjer je s velikost polobsega, $s = (a+b+c)/2$. Napiši program, ki uporabnika vpraša po dolžinah stranic in izpiše (njegovo) ploščino.

Program naj popazi na nepravilne podatke. Če uporabnik zatrdi, da ima trikotnik stranice 1, 2 in 5 (takšnega trikotnika – razmisli – pač ni), naj ga program pozove k resnosti.

Če nalogo rešuješ s funkcijo, naj se imenuje `ploscina_trikotnika`, prejme naj dolžine stranic in ne pazi na nič. Če so podatki čudni, naj se pač sesuje.

Pogoji in zanke

5. Vaja iz poštevanke

Napiši program, ki uporabniku zastavi vprašanje iz poštevanke. Uporabnik vpiše odgovor in program odgovori »Pravilno.« ali »Napačno.«

```
3 krat 10
Odgovor? 29
Napačno.
```

Namig: »3 krat 10« smo izpisali s `print`, »Odgovor?« pa z `input`. »29« je vtipkal uporabnik.

Naključno število med 2 in 10 dobiš s klicem `randint(2, 10)`. Da jo boš lahko uporabljal, na začetku programa napiši from random import *. Tvoj program se bo torej najbrž začel nekako takole

```
from random import *
a = randint(2, 10)
b = randint(2, 10)
```

6. Vsi po pet

V trgovini "Vsi po pet" morajo stranke vedno kupiti natanko pet izdelkov. Za blagajne zato potrebujejo programsko opremo, ki uporabnika (blagajnika) vpraša po petih cenah; ko jih le-ta vnese, program izpiše vsoto.

```
Cena izdelka: 2
Cena izdelka: 4.5
Cena izdelka: 1
Cena izdelka: 6
Cena izdelka: 3
Vsota: 16.5
```

7. Konkurenca

Konkurenca ne spi. Trgovina za vogalom se je odločila za posebno ponudbo: kupec lahko kupi toliko izdelkov, kolikor želi. Napiši program, ki blagajnika vpraša, koliko izdelkov je v košarici, nato vpraša po cenah teh izdelkov in na koncu izpiše vsoto.

```
Število izdelkov: 3
Cena izdelka: 2
Cena izdelka: 4.5
Cena izdelka: 1.25
Vsota: 7.75
```

8. Top-shop

Modro vodstvo tretje trgovine za drugim vogalom je analiziralo poslovanje druge trgovine in odkrilo, da ima dolge vrste na blagajnah, to pa zato, ker morajo blagajniki prešteti izdelke,

preden lahko začnejo vnašati njihove cene. Zase je naročilo nov program, ki ne vpraša po številu izdelkov, temveč sprašuje po cenah toliko časa, dokler blagajnik ne vnese ničle.

```
Cena izdelka: 2
Cena izdelka: 4
Cena izdelka: 1
Cena izdelka: 0
Vsota: 7.0
```

9. Državna agencija za varstvo potrošnikov

Zaradi poplave sumljivih trgovin za vogali se je Državna agencija za varstvo potrošnikov odločila nadzorovati poprečne cene izdelkov v košaricah strank. Popravi zadnji ali predzadnji program tako, da bo izpisal tudi poprečno ceno.

```
Cena izdelka: 2
Cena izdelka: 4
Cena izdelka: 1
Cena izdelka: 0
Vsota: 7
Poprečna cena: 2.33333333333
```

Ne vznemirjaj se zaradi grdega izpisa cene; državni uradniki so natančni in želijo vedeti vse do zadnje decimalke!

10. Collatzova domneva

Vzemimo poljubno število in z njim počnimo tole: če je sodo, ga delimo z 2, če je liho, pa ga pomnožimo s 3 in prištejmo 1. To ponavljamo, dokler ne dobimo 1.

Za primer vzemimo 12. Ker je sodo, ga delimo z 2 in dobimo 6. 6 je sodo, torej ga delimo z 2 in dobimo 3. 3 je liho, torej ga množimo s 3 in prištejemo 1 – rezultat je 10. 10 je sodo, zato ga delimo z 2 in dobimo 5... Celotno zaporedje je 12, 6, 3, 10, 5, 16, 8, 4, 2, 1. Ko pridemo do 1, se ustavimo.

Napiši program, ki mu uporabnik vnese število, program pa izpiše zaporedje, ki se začne s tem številom.

Mimogrede: naloga se ukvarja z enim slavnih nerešenih matematičnih problemov. Na Wikipediji poglej stran "Collatz conjecture".

11. Benjaminovi kovanci

Nekdo (recimo mu Benjamin) igra igro na srečo, pri kateri meče kovanec. Če pade grb, izgubi, če pade cifra dobi en kovanec. Napiši program, ki odigra eno igro. Benjamin ima v začetku 5 kovancev. Z igro konča, ko ima deset kovancev ali pa ostane brez njih. Program naj po vsakem koraku igre izpiše, kaj je padlo (G ali C) in koliko kovancev ima Benjamin.

Za metanje kovanca smeš uporabiti že pripravljeno funkcijo `vrzi`, ki ne sprejema nobenih argumentov in vrne G ali C. Postavi jo na vrh svojega programa.

```
import random
def vrzi():
    return random.choice("GC")
```

Primer izpisa:

```
G 4
G 3
G 2
C 3
C 4
G 3
G 2
C 3
G 2
G 1
G 0
```

12. Tekmovanje iz poštevanke

Sestavite program za tekmovanje iz poštevanke, ki poteka takole. Dva tekmovalca si zastavljata račune tako, da eden vpiše dva faktorja, drugi mora povedati produkt. Nato se zamenjata. Tekmovalec dobi točko, če ugane pravilni zmnožek. Po vsakem krogu program izpiše trenutni rezultat. Igre je konec, ko eden od tekmovalcev vodi za več kot dve točki. Program mu mora primerno čestitati, drugemu pa povedati, kar mu gre.

```
Tekmovalec 1, prvi faktor? 4
Tekmovalec 1, drugi faktor? 2
Tekmovalec 2, produkt? 8

Tekmovalec 2, prvi faktor? 8
Tekmovalec 2, drugi faktor? 4
Tekmovalec 1, produkt? 32

Trenutni rezultat: 1 : 1

Tekmovalec 1, prvi faktor? 4
Tekmovalec 1, drugi faktor? 5
Tekmovalec 2, produkt? 21
```

(in tako naprej, do)

```
Tekmovalec 2, prvi faktor? 3
Tekmovalec 2, drugi faktor? 3
Tekmovalec 1, produkt? 8

Trenutni rezultat: 2 : 4

Bravo drugi! Prvi, cvek!
```

13. Števke

Napiši program, ki prebere število in izpiše njegove števke v poljubnem vrstnem redu. Če uporabnik vpiše 127476, bi lahko program izpisal, recimo

6
7
4
7
2
1

Predpostaviti smeš, da je število pozitivno.

Namig: v resnici bo najlažje, če izpisuje v tem vrstnem redu. Zadnjo števko dobiš tako, da izračunaš ostanek po deljenju z 10. Po tem število deliš z 10. Vse skupaj ponavljaš, dokler ne dobiš 0.

14. Obrnjena števila

Napiši program, ki obrne podano število. Če uporabnik vpiše 8732, naj računalnik odgovori 2378. (Tole je sicer bolj vaja iz matematike kot iz programiranja.)

Če vadiš funkcije, napiši funkcijo `obrni_stevilo(n)`, ki vrne obrnjeno število.

15. Kalkulator

Če vadiš funkcije, napiši funkcijo `calc(v1, v2, oper)`, ki deluje kot preprost kalkulator. Sprejme naj dve števili (`v1` in `v2`) in operacijo `(kot niz '*', '+' ali '-')` ter vrne rezultat. Če jih še ne, nalogo preskoči. :)

Zanke prek seznamov in nizov

16. Vsota elementov seznama

Napiši program, ki izpiše vsoto elementov seznama. Seznam je lahko podan kar na začetku programa. Prva vrstica programa je torej lahko, na primer,

```
s = [5, 8, 3, 6, 0, 1]
```

Če boš nalogo sprogramiral v obliki funkcije, naj se le-ta imenuje `vsota(s)`.

17. Ajavost nizov

Napiši program, ki mu uporabnik vpiše niz in računalnik mu pove, koliko črk "a" je v njem.

Če nalogo rešuješ v obliki funkcije, naj se ta imenuje `stevilo_ajev(beseda)`. Sprogramiraj tudi `stevilo_znakov(beseda, znak)`, ki ne vrne števila a-jev temveč število podanih znakov. Končno, pobriši funkcijo `stevilo_ajev` in jo napiši na novo, tako da bo le poklicala `stevilo_znakov`.

18. Največji element

Napiši program, ki izpiše največji element seznama. Seznam je spet lahko podan kar v programu; ni potrebno, da ga vnaša uporabnik.

Namig: podobno, kot si si prej zapomnil vsoto dosedanjih elementov, si moraš zdaj zapomniti največji element doslej. Pri računanju vsote si za začetno vrednost vsote seveda izbral 0. Tu 0 ni nujno dobra izbira, saj je največji element morda negativen. Kot začetno vrednost lahko uporabiš `None` ali pa vzameš prvi element seznama; dobiš ga z `s[0]`.

Če nalogo rešuješ v obliki funkcije, naj bo to funkcija `najvecji(s)`.

19. Največji absolutist

Napiši program, ki vrne največje število po absolutni vrednosti (-8 je po absolutni vrednosti večje od -2 in od 5). Če programiraš funkcijo, naj se imenuje `najvecji_abs(s)`.

20. Najmanjši pozitivist

Napiši program, ki vrne najmanjše pozitivno število v seznamu. Negativna števila in 0 naj prezira. Če v seznamu ni pozitivnih števil, naj vrne `None`.

Če pišeš funkcijo, naj se imenuje `najmanjsi_poz(s)`.

Podobnih nalog, kot so te, si lahko izmisliš še kolikor hočeš.

21. Najdaljša beseda

Napiši funkcijo, ki prejme niz in vrne najdaljšo besedo v njem. Za niz "an ban pet podgan" naj vrne besedo "podgan". Niz `s` razbiješ na seznam besed tako, da pokličeš `besede = s.split()`.

Če reč napišeš kot funkcijo, naj se imenuje `najdaljsa_beseda(s)`.

22. Poprečje

Napiši program (ali pa funkcijo `poprecje(s)`), ki kot prejme seznam tež skupine državljanov in izračuna poprečno težo. Če je seznam prazen, je poprečna teža 0.

23. Poprečje brez skrajnežev

Napiši funkcijo `poprecje_brez(s)`, ki prejme podoben seznam kot v prejšnji nalogi, vendar računa poprečje *brez najtežjega in najlažjega*. Če si mesto najtežjega ali najlažjega deli več ljudi, naj odstrani le po enega.

24. Bomboni

Imamo gručo otrok in v seznamu je zapisano, koliko bombonov ima kateri od njih. Ker ne želimo prepirov, joka in izsiljevanja, bi radi poskrbeli, da bodo imeli vsi otroci enako bombonov. Ker bombonov seveda ne moremo jemati (prepir, jok, izsiljevanje in te reči), je naša naloga napisati funkcijo `bomboni(s)`, ki kot argument dobi seznam s številom bombonov, ki jih imajo otroci, kot rezultat pa vrne, koliko bombonov bo potrebno še razdeliti, da jih bodo imeli vsi otroci toliko, kolikor jih ima ta, ki jih ima največ.

```
>>> bomboni([5, 8, 6, 4])
9
```

Največ bombonov ima drugi otrok, 8. Torej bomo dali prvemu 3, tretjemu 2 in četrtemu 4, da jih bodo imeli vsi po 8, tako kot drugi otrok. Skupaj bomo torej razdelili 3+2+4=9 bombonov, zato funkcija vrne 9.

25. Vsaj eno liho

Napiši funkcijo `vsaj_eno_liho(s)`, ki prejme seznam števil in vrne `True`, če je med njimi vsaj eno liho, in `False`, če ni.

26. Sama liha

Napiši funkcijo `sama_liha(s)`, ki prejme seznam števil ter vrne `True`, če so vsa liha, in `False`, če niso. Če je seznam prazen, naj funkcija vrne True, saj v njem ni ne-lihih števil.

27. Blagajna

Zaporedje dogodkov na blagajni v trgovini lahko predstavimo z zaporedjem plusov in minusov: plus naj pomeni, da se je v vrsto postavila nova stranka in minus, da je stranka plačala in šla. Tako bi zaporedje `"++-++---"` pomenilo, da je najprej prišla ena stranka, nato še ena, nato je bila ena postrežena, nato sta dve prišli in potem so bile tri postrežene.

Napiši funkcijo `blagajna(s)`, ki kot argument prejme niz, kakršen je gornji, kot rezultat pa vrne najdaljšo dolžino vrste. Predpostaviti smeš, da niz vsebuje le pluse in minuse.

```
>>> blagajna("++-++---")
3
>>> blagajna("+++++-----")
5
>>> blagajna("+-+-+-+-+-+-")
1
```

28. Preobremenjeni čolni

Recimo, da na prvi čoln naložimo tri tovore, težke 4, 2 in 4 enote, na drugega en tovor, težak 10 enot in na tretjega tri tovore težke eno enoto. Tako obremenitev bi lahko opisali s seznamom seznamov (uh, uh!) `[[4, 2, 4], [10], [1, 1, 1]`.

Napiši funkcijo `ni_preobremenjenih(tovori, nosilnost)`, ki prejme seznam v takšni obliki in nosilnost čolnov (ta je za vse enaka). Funkcija naj vrne `True`, če ni nobeden od čolnov preobremenjen.

```
>>> ni_preobremenjenih([[4, 2, 4], [10], [1, 1, 1]], 11)
True
>>> ni_preobremenjenih([[4, 5, 4], [10], [1, 1, 1]], 11)
False
```

V drugem primeru je preobremenjen prvi čoln, saj nameravamo nanj naložiti za 4 + 5 + 4 = 13 eno tovora, kar je več kot dovoljena nosilnost 11.

Zanke prek številskih intervalov

29. Delitelji

Napiši program, ki izpiše vse delitelje podanega števila.

```
Vpiši število: 12345
1
3
5
15
823
2469
4115
12345
```

30. Praštevilo

Napiši funkcijo `prastevilo(n)`, ki vrne `True`, če je podano število praštevilo in `False`, če ni.

31. Vsota deliteljev

Napiši funkcijo `vsota_deliteljev(n)`, ki vrne vsoto deliteljev podanega števila. Med delitelji naj bo tudi 1, ne pa število samo. Vsota deliteljev 12 je tako 1 + 2 + 3 + 4 + 6 = 16.

32. Popolno število

Število je popolno, če je enako vsoti svojih deliteljev (razen samega sebe). Primer popolnega števila je 28, ki ga delijo 1, 2, 4, 7, in 14: če jih seštejemo, spet dobimo 28.

Napiši funkcijo `popolno(n)`, ki vrne `True`, če je podano število popolno in `False`, če ni.

Pomagaš si lahko s funkcijo iz prejšnje naloge.

33. Vsa popolna števila

Napiši program, ki izpiše vsa popolna števila manjša od 1000.

Pomagaš si lahko s funkcijo iz prejšnje naloge.

34. Prijateljska števila

220 in 284 sta prijateljski števili. Delitelji 220 so 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 in 110. Če jih seštejemo, dobimo 284. Delitelji 284 pa so 1, 2, 4, 71 in 142. Vsota teh števil pa je spet 220. Napiši funkcijo `prijatelj(n)`, ki vrne prijatelja podanega števila n. Če število nima prijatelja (ker vsota "prijateljevih" deliteljev ni spet enaka številu `n`), naj vrne `None`.

Primer: `prijatelj(220)` naj vrne `284`, `prijatelj(284)` naj vrne `220`, `prijatelj(222)` naj vrne `None`. Popolna števila so prijatelji sami sebi; `prijatelj(6)` naj vrne `6`.

35. Vsebuje 7

Napiši funkcijo `vsebuje_7(n)`, ki vrne `True`, če število vsebuje števko 7.

36. Poštevanka števila 7

Poštevanko števila 7 se igra tako, da igralci, ki sedijo v krogu, kvadratu ali kakem drugem primernem (po potrebi nepravilnem, a po možnosti konveksnem) poligonu po vrsti govorijo števila od ena do neskončno, pri čemer morajo namesto vseh števil, ki so deljiva s 7 ali pa vsebujejo števko 7, reči BUM. Igralec, ki se zmoti, izpade in štetje se začne od začetka. Igra torej teče tako:

1 2 3 4 5 6 BUM 8 9 10 11 12 13 BUM 15 16 BUM 18 19 20 BUM 22 23 24 25 26 BUM BUM 29 ...

Napiši funkcijo `postevanka_7(n)`, ki izpiše tole zaporedje od 1 do `n`. Ne da bi se zmotila. Izpisuje lahko v eni vrstici, tako kot zgoraj, če znaš. Sicer pa naj piše številke eno pod drugo.

Nalogo pokusi rešiti tako, da si pomagaš s funkcijo iz prejšnje naloge, potem pa še tako, da te funkcije ne uporabiš.

37. Fibonaccijevo zaporedje

Fibonaccijevo zaporedje se začne s številoma 1, 1, vsak naslednji člen pa dobimo tako, da seštejemo prejšnja dva. 1 in 1 je 2, 1 in 2 je 3, 2 in 3 je 5, 3 in 5 je 8 in tako naprej. Zaporedje se tako začne z 1 1 2 3 5 8 13 21 34 55.

Napiši program, ki izpiše prvih 20 členov zaporedja.

38. Evklidov algoritem

Napiši funkcijo `gcd(a, b)`, ki z Evklidovim algoritmom poišče največji skupni delitelj podanih števil `a` in `b`. Opis algoritma – če si ga pozabil – poišči na spletu. (Vendar ne prepiši skupaj z algoritmom še programa, razen če želiš vaditi tipkanje ali kopipejstanje, ne programiranja!)

Namig: Evklidov algoritem v vsakem koraku zamenja večje in manjše število z manjšim številom in ostankom večjega po deljenju z manjšim. Program je podoben prejšnjemu, vendar ravno prav drugačen, da ju je zanimivo opazovati skupaj.

Kratica gcd pomeni *greatest common divisor*, največji skupni delitelj.

Zanke prek več reči hkrati

39. Indeks telesne teže

Imejmo seznam trojk ime osebe, teža, višina, na primer:

```
podatki = [
    ["Ana", 55, 165],
    ["Berta", 60, 153],
]
```

Napiši program, ki izpiše imena oseb in njihove indekse telesne teže. Dobimo jo tako, da težo delimo s kvadratom višine v metrih.

40. Seštete trojke

Napiši funkcijo, ki prejme seznam trojk in vrne `True`, če za vse trojke velja, da je tretji element vsota prvih dveh; v nasprotnem primeru vrne `False`.

Tako bo, na primer, za seznam `[(3, 5, 8), (2, 6, 9), (1, 1, 2), (10, 5, 15)]` vrnila `False`, ker je v njem nepravilna trojka: 2 + 6 ni enako 9.

Kako pa bi rešil nalogo, če vrstni red elementov v trojki ne bi bil določen in bi bila pravilna tudi, recimo, trojka (3, 8, 5), saj lahko enega od elementov dobimo kot vsoto ostalih dveh? Napiši funkcijo `neurejene_trojke`, ki zna tudi to.

41. Skalarni produkt

Napiši funkcijo `skalarni(v, w)`, ki vrne skalarni produkt dveh vektorjev. Vektorje lahko predstavimo s seznami ali terkami. Skalarni produkt vektorjev `(1, 2, 3)` in `(5, 4, 6)` je 1*5 + 2*4 + 3*6 = 31. Klic `skalarni((1, 2, 3), (5, 4, 6))` mora torej vrniti `31`. Vektorji so lahko poljubno dimenzionalni.

42. Ujemanja črk

Napiši funkcijo `st_ujemanj(b1, b2)`, ki prejme dve besedi in vrne število istoležnih črk, v katerih se ujemata. TRAVNIK in PRAVNIK se v šestih, MLEKO in MREŽA v dveh, OČALA in MARKO pa v nobeni (obe imata sicer O in A, vendar ne na istih mestih). Besedi nista nujno enako dolgi; PAV in KRVAVICA se ujemata v eni črki, TRAVNIK in RAVNIK pa v nobeni.

43. Vzorec besede

Recimo, da rešujemo križanko. Imamo besedo s petimi črkami, pri čemer vemo, da je prva M, druga L in peta O. Takšen vzorec bomo predstavili z nizom `"ML..O"`. Iskana beseda bi lahko bila, recimo, MLEKO.

Napiši funkcijo `se_ujema(beseda, vzorec)`, ki vrne `True`, če se podana beseda ujema z vzorcem.

44. Prva beseda

Napiši funkcijo `prva_beseda(besede, vzorec)`, ki dobi seznam besed in vzorec, kakršne smo videli v prejšnji nalogi. Vrniti mora prvo besedo iz seznama, ki se ujema z vzorcem. Če se ne ujema nobena, vrne `None`.

45. Paralelni skoki

V Planici so zgradili novo, dvojno skakalnico, s katero lahko izvajajo meddržavna tekmovanja v paralelnih skokih. Stvar je preprosta: skakalci se paroma odganjajo z dveh skakalnic, eni z ene, drugi z druge. Država, katere skakalec je skočil dlje, dobi točko; če sta skoka enako dolga, dobi vsaka država polovico točke. Zmaga država, ki dobi več točk.

Napiši funkcijo, ki dobi seznama z dolžinami skokov. Če je zmagala prva država, naj funkcija vrne 1, sicer 2. Če sta državi izenačeni, naj funkcija ne vrne ničesar (`None`).

```
>>> paralelni_skoki([153, 141, 152, 160, 135], [148, 148, 148, 148, 148])
1
```

Finci so zmagali, ker so bili boljši v treh skokih, (fascinantno usklajeni) Danci pa le v dveh.

46. Mesto največjega elementa

Napiši funkcijo `arg_max(s)`, ki vrne, kje v seznamu se nahaja njegov največji element. Če imamo seznam `[5, 1, 4, 8, 2, 3]`, naj vrne `3`, saj je največji element, `8`, na tretjem mestu (če začnemo, kot se spodobi, šteti z `0`). Če je v seznamu več enakih največjih števil, naj se izpiše mesto, na katerem se nahaja prvo. Tudi za seznam `[5, 1, 4, 8, 2, 3, 8, 8, 8]` mora program izpisati `3`. Če je seznam prazen, naj funkcija vrne `None`.

(Ime, `arg_max` je običajno ime za tovrstne funkcije.)

47. Olimpijske medalje

Tabela kaže razvrstitev desetih držav glede na število medalj na olimpijskih igrah v letih 2016 in 2012. Vidimo, da so tri države na tej lestvici napredovale, tri pa nazadovale.

Napiši funkcijo `napredek(s)`, ki kot argument prejme seznam števil v drugem stolpcu (npr. `[1, 3, 2, 4, 6, 10, 7, 5, 9, 8]`), kot rezultat pa vrne par (terko) s števili, ki povesta, koliko držav je na lestvici napredovalo in koliko nazadovalo.

Funkcija mora delovati za poljubno dolge sezname, ne le za deset držav.

letošnje	prejšnje	
1	1	United States
2	3 ^	Great Britain
3	2 v	China
4	4	Russia
5	6 ^	Germany
6	10 ^	Japan
7	7	France
8	5 v	South Korea
9	9	Italy
10	8 v	Australia

48. Vstavi teže

Imamo seznam imen; ženska imena se vedno končajo s črko `"a"`, moška pa nikoli. Vsi moški so nam zaupali svoje teže. Napišite funkcijo `vstavi_teze(osebe, teze)`, ki prejme tadva seznama in imena moških zamenja z njihovimi težami (po vrsti), imena žensk pa pusti pri miru. Funkcija naj ne vrača ničesar – spreminjati mora podani seznam.

```
>>> imena = ["Adam", "Eva", "Kajn", "Abel"]
>>> teze = [87, 86, 75]
>>> vstavi_teze(imena, teze)
[87, "Eva", 87, 75]
```

49. Primerjanje seznamov

Napiši funkcijo `primerjaj(s, t)`, ki primerja dva (neprazna) seznama, takole:

- če sta seznama enaka, naj vrne `0`;
- če sta enako dolga in so vsi elementi `s` manjši ali enaki od istoležnih elementov `t`, naj vrne `-1`;
- če sta enako dolga in so vsi elementi `s` večji ali enaki od istoležnih elementov `t`, naj vrne `1`;
- sicer vrne `0`.

```
>>> primerjaj([1, 2, 3, 4], [2, 3, 4, 5])
-1
>>> primerjaj([2, 3, 4, 5], [1, 2, 0, 0])
1
>>> primerjaj([1, 2, 3], [4, 5, 6, 7])
0
>>> primerjaj([1, 0], [0, 1])
0
```

Indeksiranje, seznami, nizi

50. Spol iz EMŠO

Številka EMŠO je sestavljena iz trinajst števk v obliki DDMMLLL50NNNX, pri čemer je DDMMLLL rojstni datum, 50 je koda registra (EMŠO je nastala v času Jugoslavije in 50 je bila koda za Slovenijo), NNN je zaporedna številka in X kontrolna številka. Trimestna številka, NNN, je med 000 in 499 za moške ter med 500 in 999 za ženske.

Napiši funkcijo `je_zenska(emso)`, ki za številko EMŠO, ki jo podamo kot niz, vrne `True`, če pripada ženski in `False`, če moškemu.

51. Pravilnost EMŠO

Zadnja števka v EMŠO je kontrolna. Izračuna se takole: prvo števko EMŠO pomnožimo s 7, drugo s šest, tretjo s pet in tako naprej, do šeste, ki jo pomnožimo z 2. Sedmo spet pomnožimo s 7, osmo s 6, deveto s 5 in tako do dvanajste, ki jo pomnožimo z 2. Za zadnjo, trinajsto števko, velja tole: če jo prištejemo h gornji vsoti, dobimo število, ki je deljivo z 11.

Napiši funkcijo `preveri_emso(emso)`, ki preveri pravilnost podane številke EMŠO.

52. Starost iz EMŠO

Napiši funkcijo `starost(emso)`, ki kot argument prejme EMŠO osebe in kot rezultat vrne njeno dopolnjeno starost v letih. Predpostavite, da smo 26. januarja 2010, torej je nekdo, ki je bil rojen 20. januarja 2000 (ali celo 26. januarja 2000) star 10 let, nekdo, ki je rojen 10. februarja 2000 pa samo 9 let, ker letos še ni praznoval rojstnega dne.

Rojstni datum osebe pove prvih sedem števk EMŠO. EMŠO osebe, rojene 10. 5. 1983, bi se začel z 1005983, prvih sedem števk EMŠO osebe, rojene 2. 3. 2001, pa bi bilo 0203001. Ali je oseba rojena leta 1xxx ali 2xxx, sklepamo po stotici.

```
>>> starost("2601971500123")
39
>>> starost("2001971500123")
39
>>> starost("2002971500125")
38
```

53. Domine

Vrsta domin je podana s seznamom parov (terk), na primer `[(3, 6), (6, 6), (6, 1), (1, 0)]` ali `[(3, 6), (6, 6), (2, 3)]`. Napišite funkcijo `domine(s)`, ki prejme takšen seznam in pove, ali so domine pravilno zložene. Za prvega od gornjih seznamov mora vrniti `True` in za drugega `False`.

54. Dan v letu

Napiši funkcijo `dan_v_letu(dan, mesec)`, ki prejme datum v letu 2010 in vrne zaporedno številko dneva v letu. Tako je, na primer, 10. februar 41. dan v letu.

Leto 2010 seveda ni bilo prestopno.

```
>>> dan_v_letu(26, 1)
26
>>> dan_v_letu(10, 2)
41
>>> dan_v_letu(31, 12)
365
```

55. Nepadajoči seznam

Napiši funkcijo `nepadajoc(s)`, ki za podani seznam pove, ali je nepadajoč, torej, ali je vsak element enak ali večji od svojega predhodnika.

56. Mesta črke

Napiši funkcijo `mesta_crke(beseda, crka)`, ki vrne seznam mest v besedi, na katerih nastopa podana črka.

```
>>> mesta_crke("PONUDNIK","N")
[2, 5]
>>> mesta_crke("RABARBARA", "R")
[0, 4, 7]
>>> mesta_crke("AMFITEATER", "O")
[]
```

57. Multiplikativni range

Napiši funkcijo, ki vrne seznam, kjer je vsako naslednje število za podani faktor večje od prejšnjega. Npr., v seznamu `[1,2,4,8,16]` je vsako naslednje število dvakrat večje od prejšnjega. Argumenti funkcije naj bodo začetno število, faktor in dolžina seznama.

```
>>> mrange(7, 4, 5)
[7, 28, 112, 448, 1792]
```

58. Sumljive besede

Napiši funkcijo, ki vrne seznam vseh sumljivih besed v danem nizu. Beseda je sumljiva, če vsebuje tako črko `u` kot črko `a`.

```
>>> sumljive('Muha pa je rekla: "Tale juha se je pa res prilegla, najlepsa
huala," in odletela.')
['Muha', 'juha', 'huala,"']
```

59. Kockarji

N kockarjev eden za drugim meče kocko in mete beleži v seznam. Če bi prvi kockar vrgel 2, drugi 3, tretji 5, prvi 4, drugi 3, tretji 6, bi to zapisali kot `[2, 3, 5, 4, 3, 6]`.

Napišite funkcijo `kockarji(s, n)`, ki prejme seznam in število kockarjev, kot rezultat pa vrne zaporedno številko kockarja, ki je vrgel največ šestic. (Prvi kockar naj ima zaporedno številko 1, ne 0.) Predpostaviti smete, da v igri sodeluje vsaj en igralec in, če potrebujete, tudi, da je dolžina seznama večkratnik `n`.

```
>>> kockarji([1, 2, 6,   1, 2, 6,   1, 6, 6,   1, 2, 1], 3)
3
>>> kockarji([1, 6,   1, 6,   2, 2], 2)
2
```

60. Križanka

Napiši funkcijo `krizanka(vzorec, besede)`, ki dobi kot argument napol izpolnjeno besedo `vzorec`, v kateri so manjkajoče črke zamenjane s pikami, in seznam besed `besede`. Funkcija naj vrne seznam vseh besed, ki se ujamejo s podano besedo.

```
>>> krizanka("r.k.", ["reka", "rokav", "robot", "roka"])
['reka', 'roka']
```

Namig: lažje bo, če najprej pripraviš funkcijo, ki dobi dve besedi in pove, ali druga ustreza prvi. To funkcijo nato kliči v funkciji, ki jo zahteva naloga.

61. Sosedi

Za neko krožno ulico (hiše so razporejene v krogu) imamo seznam, ki pove koliko ljudi živi v vsaki od hiš. Napiši funkcijo `stevilo_sosedov(prebivalci)`, ki za vsako hišo pove, koliko ljudi živi v sosednjih dveh hišah. Za seznam `[1,2,0,5]` naj funkcija vrne `[7,1,7,1]`. Predpostaviti smeš, da so v vsaki krožni ulici vsaj tri hiše.

62. Glajenje

Napiši funkcijo, ki izračuna tekoče poprečje danega zaporedja. Kot argument naj prejme seznam števil, kot rezultat pa vrne seznam poprečij po štirih zaporednih elementov.

Tekoče poprečje seznama `[3, 5, 8, 0, 7, -3, 12, 0, -5, 5]` je `[4, 5, 3, 4, 4, 1, 3]`: poprečje elementov 3, 5, 8, 0 je 4, poprečje 5, 8, 0, 7 je 5, poprečje 8, 0, 7, -3 je 3 ...

63. An ban pet podgan

Napišite funkcijo, ki za dani seznam poišče zmagovalca v izštevanki "An ban pet podgan". V vsakem odštevanju izpade tisti, na katerega pokažemo ob besedi "ven". Zmagovalec igre je tisti, ki na koncu edini ostane neizločen.

Za preverjanje: če imamo osebe `["Maja", "Janja", "Sabina", "Ina", "Jasna"]` je zmagovalec Jasna. Če izštevamo osebe `["Maja", "Janja", "Sabina"]`, zmaga Maja.

Za tiste, ki ste že malo pozabili; celotna izštevanka se glasi: "an ban, pet podgan, štiri miši, v'uh me piši, vija, vaja, ven."

64. Največji skupni delitelj seznama

Če bi rad odkril največji skupni delitelj štirih števil, recimo a, b, c in d, lahko to storiš tako, da poiščeš največji skupni delitelj a in b; imenujmo ga ab. Nato poiščeš največji skupni delitelj ab in c; imenujmo ga abc. Nazadnje poiščeš največji skupni delitelj abc in d; to je največji skupni delitelj vseh štirih.

Napiši funkcijo `skupni_delitelj(s)`, ki kot argument prejme seznam števil in kot rezultat vrne njihov največji skupni delitelj.

65. Oškodovani otroci

Mama deli bombone šestim otrokom, ki jih je oštevilčila s številkami od 0 do 5. Zaporedje dajanja bombonov je shranila v seznam. Tako, recimo, `[4, 0, 2, 3, 2, 0, 3, 4, 4, 4]` pomeni, da je najprej dala bombon četrtemu otroku, nato ničtemu, potem drugemu...

Napiši funkcijo, ki dobi takšen seznam otrok in vrne `True`, če so dobili bombone vsi otroci, in `False`, če je kateri ostal brez. V gornjem primeru mora vrniti `False`, ker otrok 1 (pa tudi 5) ni dobil bombona.

66. Lomljenje čokolade

Imamo kvadratno čokolado iz n x n koščkov. Kako jo lomimo, pove niz, katerega prvi znak je stran, na kateri jo lomimo, preostali znaki so število odlomljenih stolpcev oz. vrstic:

- `"<3"` pomeni, da odlomimo tri stolpce z leve,
- `">12"` pomeni, da odlomimo 12 stolpcev z desne,
- `"^1"` pomeni, da odlomimo zgornjo vrstico,
- `"v5"` pomeni, da odlomimo spodnjih 5 vrstic.

Če poskušamo odlomiti več, kot je možno (imamo pet vrstic in uporabimo `"v7"`), pač odlomimo, kolikor gre (samo pet vrstic)

Napiši funkcijo `cokolada(n, odlomi)`, ki prejme velikost čokolade (`n`) in seznam lomljenj (odlomi, na primer `["<3", ">2", ">1", "v12", "<7"]`. Funkcija naj pove, koliko kvadratkov čokolade je še ostalo. Klic `cokolada(10, ["<3", "v5"])` vrne 35, saj je ostalo 7 stolpcev in 5 vrstic.

67. Razcep na praštevila

Kot učijo že osnovnošolce, lahko vsako število razcepimo v produkt praštevil. Tako je 252 enako $2^2\ 3^2 7^1$ in 1944 je $2^3 3^5$. Napiši funkcijo, ki kot argument prejme število in kot rezultat vrne seznam terk, ki predstavljajo osnovo in potenco. Za gornja primera mora izgledati takole:

```
>>> razcep(252)
[(2, 2), (3, 2), (7, 1)]
>>> razcep(1944)
[(2, 3), (3, 5)]
```

Da povadimo pisanje funkcij, naj bo rešitev sestavljena iz več funkcij.

1. Napiši funkcijo `prastevilo(n)`, ki ugotovi, ali je dano število `n` praštevilo.

```
>>> prastevilo(42)
False
>>> prastevilo(43)
True
```

2. Napiši funkcijo `prastevila(n)`, ki vrne vsa praštevila med (vključno!) `2` in `n`. Funkcija naj deluje tako, da za vsako števila od `2` do `n` pokliče gornjo funkcijo `prastevilo(n)` in ga, če le-ta pravi, da gre za praštevilo, doda na seznam.

```
>>> prastevila(100)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
73, 79, 83, 89, 97]
>>> prastevila(43)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43]
```

3. Napiši funkcijo `deljivost(n, i)`, ki pove, kolikokrat je `n` deljiv z `i`. (Namig: *dokler* je `n` deljiv z `i`, ga deli z `i` in sproti štej, kolikokrat si ga delil.)

```
>>> deljivost(1944, 2)
3
>>> deljivost(1944, 3)
5
```

4. Napiši funkcijo `razcep(n)`, ki naredi, kar zahteva naloga, pri čemer kliče funkciji `prastevila` in `deljivost`: za vsa praštevila med 2 in n ugotovi, kolikokrat delijo podano število in če ga delijo vsaj enkrat (torej, če ga delijo), to doda v seznam. (Namig: če želiš dodati v seznam `t` terko `(1, 2)`, rečeš `t.append((1, 2))` in ne `t.append(1, 2)`.)

Rešitev ne bo preveč učinkovita, bo pa poučna, saj zahteva pravilno pisanje funkcij.

68. Pari števil

Napiši funkcijo `pari()`, ki vrne seznam parov števil med `1` in `1000`, za katera velja, da imata različno število mest, vendar je vsota njunih števk enaka. Vsak par naj se pojavi le enkrat; v seznamu naj bo torej, recimo, par `(76, 526)` ali `(526, 76)`, ne pa oba.

Primer: prvih deset elementov seznama je lahko, recimo, takšnih: `(76, 526)`, `(88, 754)`, `(8, 134)`, `(27, 153)`, `(95, 617)`, `(67, 922)`, `(52, 7)`, `(73, 541)`, `(69, 465)`, `(96, 726)`.

Par `(76, 256)` je na seznamu, ker `7+6 = 5+2+6`. Pač pa na seznamu ni para `(760, 526)`, saj imata števili enako število mest.

69. Ploščina poligona

Ploščino poligona, katerega oglišča so podana s koordinatami (x_1, y_1), (x_2, y_2), … (x_n, y_n), izračunamo po formuli

$$p = \frac{1}{2}\left|(x_1 y_2 - x_2 y_1) + (x_2 y_3 - x_3 y_2) + \ldots + (x_{n-1} y_n - x_n y_{n-1}) + (x_n y_1 - x_1 y_n)\right| .$$

Napiši funkcijo `ploscina(o)`, ki kot argument prejme seznam oglišč poligona in kot rezultat vrne njegovo ploščino. Pazi na zadnji člen!

```
>>> ploscina([(0, 0), (1, 0), (1, 1), (0, 1)])
1.0
>>> ploscina([(0, 0), (1, 0), (1, 1)])
0.5
>>> ploscina([(0, 0), (1, 0), (2, .5), (1, 1), (1, 2), (.5, 1), (0, 2)])
2.0
```

70. Sodost čebel

Pri procesu izdelave medu iz nektarja je pomembno, da ločimo čebele, ki so obrale liho število cvetov od čebel, ki so obrale sodo število cvetov.

Podan je seznam, ki vsebuje terke `(ime_cebele, stevilo_obranih_cvetov)`. Napišite funkcijo `loci(cebele)`, ki prejme takšen seznam in vrne dva seznama. V prvem naj bodo imena čebel, ki so obrale sodo število cvetov, v drugem pa imena čebel, ki so obrale liho število cvetov. Za seznam `[('Ana', 5), ('Berta', 7), ('Cilka', 2)]` naj funkcija vrne seznama `['Cilka']` in `['Ana', 'Berta']`. Čebele v obeh seznamih naj bodo urejene enako kot v prvotnem seznamu.

71. Sodi – lihi

Napiši funkcijo `sodi_lihi(sez)`, ki kot argument prejme seznam in vrne vrednost resnično, če se v seznamu izmenjujejo liha in soda števila, ter neresnično, če si kdaj zaporedoma sledita dve sodi oziroma dve lihi števili.

```
>>> sodi_lihi([3, 4, 5, 6, 3, 2, 7, 12])
True
>>> sodi_lihi([3, 4, 5, 5, 6, 3, 2, 7, 12])
False
>>> sodi_lihi([3, 4])
True
>>> sodi_lihi([3, 3])
False
>>> sodi_lihi([3])
True
>>> sodi_lihi([])
True
```

72. Najprej lihi

Napišite funkcijo `najprej_lihi(s)`, ki kot argument prejme seznam števil `s`. Njihov vrstni red naj spremeni tako, da bodo v njem najprej liha, nato soda števila – vsaka zase v enakem vrstnem redu, kot so bila prej. Funkcija ne sme vračati ničesar.

```
>>> s = [5, 8, 4, 17, 13, 10, 9]
>>> najprej_lihi(s)
>>> s
[5, 17, 13, 9, 8, 4, 10]
```

5, 17, 13 in 9 so v enakem vrstnem redu kot prej; soda tudi.

73. Sodo in liho in sodo in liho

Napišite funkcijo `soda_liha(s)`, ki kot argument prejme seznam števil `s`. Vrstni red števil naj spremeni tako, da se bodo v njem izmenjevala soda in liha števila, začenši s sodim. Oboja naj bodo v enakem vrstnem redu, kot so bila prej. Če je sodih več kot lihih ali obratno, naj odvečna števila odstrani.

```
>>> s = [5, 8, 4, 2, 8, 17, 13, 10, 2, 4, 6, 8, 9]
>>> soda_liha(s)
>>> s
[8, 5, 4, 17, 2, 13, 8, 9]
```

Po klicu funkcije so soda števila v enakem zaporedju kot prej, (8, 4, 2, 8), mednje pa so prepletena liha v enakem zaporedju kot prej (5, 17, 13, 9). Odvečna soda števila (10, 2, 4, 6, 8) so izpuščena.

74. Plus - minus - plus

Napiši funkcijo `alterniraj(s)`, ki prejme seznam števil in ga spremeni tako, da za vsakim pozitivnim številom odstrani vsa števila do naslednjega negativnega števila, in za vsakim negativnim vsa števila do naslednjega pozitivnega. Funkcija naj spremeni *podani* seznam in ne vrne ničesar.

Z drugimi besedami, funkcija obdrži prvi element vsakega zaporedja pozitivnih ali negativnih števil.

Seznam `[3, 4, -1, 1, -5, -2, -1, 7, -8]` tako spremeni v `[3, -1, 1, -5, 7, -8]`.

75. Pecivo

Tole je naloga, ki so jo na nekem tekmovanju reševali od četrtega razreda OŠ naprej.

Suzana in Aljaž sta odprla pekarno. Suzana peče pecivo v obliki črk A, B in O. Vedno speče vse tri oblike in jih obesi tako, da najprej natakne A, nato B, nato O. Aljaž jih medtem prodaja (vendar ne proda nobene v tem času, ko jih

Suzana natika). Suzana jih peče hitreje, kot se prodajajo.

Če je pekarna videti, kot kaže slika: najmanj koliko kosov peciva sta prodala?

***Rešitev:** devet kosov.*

Kako dobimo takšno zaporedje? Napišemo zaporedje, ki vsebuje toliko Ajev, kolikor jih je na sliki: ABOABOABOABOABOABO. Nato prečrtamo, česar ni na sliki: A~~BO~~A~~BO~~AB~~O~~A~~BO~~ABOA~~BO~~. Preštejemo prečrtane črke: devet jih je.

Napiši funkcijo `pecivo(s)`, ki prejme niz `s`, na primer `AAABAABOAABO` in vrne najmanjše možno število prodanih prest (v gornjem primeru 9).

Naloge raje ne rešuj natančno tako, kot predlaga gornja rešitev; z računalnikom gre preprosteje.

76. Nogavice brez para

Ker smo moški, vemo, barvno slepi, obenem pa v družini z ženo in *n* otroki ni mogoče, da bi imel vsak same enake nogavice, se lahko znajdemo tako, da na vsako nogavico prišijemo številko. Enake nogavice imajo enako številko; ker imamo več enakih nogavic, ima lahko več nogavic enako številko.

Recimo, da iz pralnega stroja potegnemo nogavice s številkami `[1, 2, 3, 2, 3, 1, 3, 1, 1, 1, 1]`. Imamo torej tri pare nogavic 1, en par nogavic 2 in en par nogavic 3 in (eh, spet!) še eno 3 brez para.

Napiši funkcijo `nogavice(s)`, ki prejme seznam številk nogavic in vrne seznam vseh številk nogavic brez para. V gornjem primeru torej vrne `[3]`. Vrstni red elementov v seznamu naj bo enak vrstnemu redu zadnjih pojavitev teh številk; za `[1, 1, 4, 1, 3, 1]` vrne `[4, 3]` in ne `[3, 4]`.

77. Presedanje

Neki dan so za okroglo mizo sedele Ana, Berta, Cilka, Dani in Ema (v tem vrstnem redu). Bile so nesrečne, zato so se drugi dan presedle v drug vrstni red: Cilka, Dani, Ema, Ana in Berta. Ni pomagalo saj so, če dobro razmislimo ... sedele enako, saj je miza okrogla in so se pravzaprav le zavrtele – vsaka je imela še vedno isto levo in desno sosedo, vir njihove nesrečnosti.

Napiši funkcijo `enaka_razporeda(razpored1, razpored2)`, ki pove, ali sta dva razporeda enaka ali ne. Razporeda sta podana s seznamom imen. Če ti pride prav, smeš predpostaviti, da imajo vsi gostje različna imena (ali pa da seznam namesto imen vsebuje Enotne matične številke občank). Če ne razumeš, zakaj bi ti ta predpostavka olajšala življenje, se ne vznemirjaj.

```
>>> r1 = ["Ana", "Berta", "Cilka", "Dani", "Ema"]
>>> enaka_razporeda(r1, ["Cilka", "Dani", "Ema", "Ana", "Berta"])
True
>>> enaka_razporeda(r1, ["Berta", "Ana", "Cilka", "Dani", "Ema"])
False
>>> enaka_razporeda(r1, ["Ana", "Berta", "Cilka"])
False
```

```
>>> enaka_razporeda(r1, ["Greta", "Helga"])
False
>>> enaka_razporeda(r1, [])
False
>>> enaka_razporeda([], [])
True
```

Slovarji in množice

78. Pokaži črke

Napiši funkcijo `pokazi_crke(beseda, crke)`, ki kot argument sprejme besedo in množico (`set`) črk. Funkcija mora vrniti besedo, v kateri so vse črke, ki ne nastopajo v množici `crke`, spremenjene v pike.

```
>>> pokazi_crke("PONUDNIK", set(["O", "N"]))
'.ON..N..'
>>> pokazi_crke("PONUDNIK", set(["O", "I", "K"]))
'.O....IK'
>>> pokazi_crke("PONUDNIK", set())
'........'
>>> pokazi_crke("PONUDNIK", set(["P", "O", "N", "I", "K", "U"]))
'PONU.NIK'
```

79. Podobna beseda

Globalna spremenljivka `besede` vsebuje seznam besed. Naj bo takšen:

```
besede = ["ana", "berta", "cilka", "dani", "ema", "fanci", "greta", "hilda"]
```

Predpostaviti smeš, da so vse besede zapisane s samimi malimi črkami.

Sestavi funkcijo `podobna(beseda)`, ki kot argument sprejme besedo in kot rezultat vrne tisto besedo iz gornjega seznama, v kateri se pojavi čim več črk, ki se pojavijo tudi v dani besedi. Vsako ujemajočo se črko štejemo le enkrat, tudi če se v obeh besedah pojavi večkrat.

Funkcija naj deluje tako, da ne razlikuje med malimi in velikimi črkami.

```
>>> podobna("merjasec")
'berta'
>>> podobna("zmaj")
'ema'
>>> podobna("Krava")
'berta'
```

Ana in krava se ujemata v eni črki, namreč črki *a* in ne v dveh, pa čeprav imata po dva *a*-ja. (Opomba: vsaka podobnost med Berto in merjascem je zgolj naključna.)

80. Število znakov

Napiši funkcijo `najraznolika(bes)`, ki kot argument prejme seznam nizov in kot rezultat vrne niz, ki ima največ *različnih* znakov. Male in velike črke upoštevaj kot iste znake - beseda "MamA" ima samo dva različna znaka. Če obstaja več besed z enakim številom različnih znakov, naj vrne prvo med njimi.

```
>>> besede = ["RABarbara", "izpit", "zmagA"]
>>> najraznolika(besede)
'izpit'
```

81. Najpogostejša beseda in črka

Napišite funkciji `najpogostejsa_beseda(s)` in `najpogostejsi_znak(s)`, ki vrneta najpogostejšo besedo in najpogostejši znak v podanem nizu `s`. V nizu `'in to in ono in to smo mi'` se največkrat pojavita beseda `'in'` in znak `' '` (presledek).

82. Samo enkrat

Napiši funkcijo `samo_enkrat(s)`, ki pove, ali se v podanem nizu `s` noben znak ne pojavi več kot enkrat.

83. Popularni rojstni dnevi

Napiši funkcijo `histogram_dni(imedat)`, ki kot argument dobi ime datoteke, v kateri je seznam številk EMŠO, v vsaki vrstici po ena. Nato izračuna, koliko ljudi je rojenih na posamezni dan v letu in to izpiše (po dnevih). Izpis mora biti v obliki, ki jo kaže primer (vključno s presledki pred in med številkami). Dneve, na katere ni rojen nihče, naj izpusti. Vrstni red dni ni pomemben.

Rojstni datum razodene prvih sedem števk EMŠO. EMŠO osebe, rojene 10. 5. 1983, bi se začela z 1005983. Prvih sedem števk EMŠO osebe, rojene 2. 3. 2001, pa bi bile 0203001. Ali je oseba rojena leta 1xxx ali 2xxx, sklepamo po stotici.

Recimo, da so v datoteki emso.txt zapisane naslednje številke.

```
1302935505313
1002968501123
1302003500231
2110987505130
1302999350538
2110912501130
```

Funkcija mora tedaj delovati takole.

```
>>> histogram_dni("emso.txt")
10. 2. 1
13. 2. 3
21.10. 2
```

84. Osrednja obveščevalna služba

Osrednja obveščevalna služba se je odločila spremljati elektronsko pošto določenih osebkov, pri čemer jo zanimajo predvsem imena, ki se pojavljajo v njej. Kot možno ime štejemo besede, ki se začnejo z veliko začetnico in nadaljujejo z malimi. Radi bi imeli program, ki bi za dano elektronsko sporočilo naštel vsa imena, ki se pojavijo v njem in število njihovih pojavitev.

Nekdo jim je že napisal program, ki vzame sporočilo in ga predela v sporočilo brez ločil. Začnemo torej s takšnim nizom:

```
msg = """Dragi Ahmed kako si kaj Upam da so otroci že zdravi Mustafa Osama in jaz
smo se šli danes malo razgledat in kaže kar dobro Abdulah pa ni mogel zraven je
šel v Pešavar prodat še tri kamele Osama sicer pravi da se mu to pred zimo ne
splača ampak saj veš kakšen je Abdulah tak je kot Harun nič jima ne dopoveš še
Osama ne Jibril me kliče moram iti oglasi se kaj na Skype tvoj Husein"""
```

Program naj izpiše nekaj takšnega:

```
Dragi 1
Ahmed 1
Upam 1
Mustafa 1
Osama 3
Abdulah 2
Pesavar 1
Harun 1
Jibril 1
Skype 1
Husein 1
```

85. Menjave

Napiši funkcijo `zamenjano(s, menjave)`, ki prejme seznam `s` in slovar `menjave`. Vrne naj nov seznam, v katerem so vsi elementi seznama, ki nastopajo kot ključi v slovarju, zamenjani s pripadajočimi vrednostmi. Elemente, ki se ne pojavijo v slovarju, pusti pri miru.

Klic `zamenjano(["Ana", "Ana", "Berta", "Ana", "Cilka"], {"Ana": "Peter", "Berta": "Ana"})` vrne `["Peter", "Peter", "Ana", "Peter", "Cilka"]`.

Funkcija `zamenjano` ne sme spremeniti podanega seznama `s`.

Poleg tega napiši podobno funkcijo `zamenjaj(s, menjave)`, ki pa ne vrne ničesar temveč ustrezno spremeni podani seznam `s`.

86. Anagrami

Napiši funkcijo, ki kot argument prejme dve besedi in pove (tako da vrne `True` ali `False`), ali sta anagrama. Besedi sta anagrama, če lahko dobimo eno besedo iz druge tako, da jima premešamo črke.

```
>>> anagram("pirat", "ripat")
True
>>> anagram("tipka", "pirat")
False
>>> anagram("tipka", "piikat")
False
```

87. Bomboniera

(Za inspiracijo za nalogo hvala kolegom s FMF!) Benjamin je našel bomboniero, v kateri je `sirina` stolpcev in `visina` vrstic bombonov. Že pred njim jo je našla in nekoliko izropala njegova sestra Ana. Benjamin se ne bo dotaknil vrstic in stolpcev, iz katerih je Ana že vzela kak

bombon. Pojedel pa bo vse ostale. Napiši funkcijo `bomboniera(sirina, visina, pojedeno)`, ki pove, koliko bombonov bo pojedel. Seznam pojedeno vsebuje pare z vodoravnimi in navpičnimi koordinatami.

Če pokličemo `bomboniera(8, 5, [(2, 1), (2, 4)])` bo Benjamin pojedel vse bombone razen tistih v drugem stolpcu ter tistih v prvi in četrti vrstici, saj je tam stikala že Ana.

88. Prafaktorji in delitelji

Napiši funkcijo `prafaktorji(n)`, ki razcepi podano število n na prafaktorje in vrne razcep v obliki slovarja. Če pokličemo `prafaktorji(1400)`, vrne slovar `{2: 3, 5: 2, 7: 1}`, saj je 1400 = $2^3 5^2 7^1$.

Nato napišite funkcijo `gcd(a, b)`, ki prejme dva slovarja, kakršna vrača prejšnja naloga, in vrne največji skupni delitelj števil, ki ju predstavljata tadva slovarja. Če pokličemo `gcd({2: 3, 5: 2, 7: 1}, {2: 2, 7: 2, 11:1})`, vrne `28`. Prvi slovar namreč predstavlja število 1400 in drugi število 2156, njun največji skupni delitelj pa je 28. Nalogo reši, ne da bi izračunal števili (npr. 1400 in 2156). Delaj s slovarjema, ki si ju dobil.

Namig: 1400 = $2^3 5^2 7^1$ in 2156 = $2^2 7^2 11^1$, zato je njun največji skupni delitelj enak $2^2 7^1$.

89. Hamilkon

Šahovski konj se premika v obliki črke L, vedno za dve polji v eni smeri in eno v drugi, npr. a1-c2 ali e5-d3. Na ta način lahko prehodi celo šahovnico tako, da začne na poljubnem polju, preskače vsa polja (na vsako polje skoči le enkrat!) in konča tam, kje je začel. Napiši funkcijo `hamilkon(s)`, ki prejme seznam polj (npr. `["g3", "h1", "f2", "d1"]`) in vrne `True`, če podani seznam vsebuje pravilni obhod in `False`, če ne.

Predpostavimo lahko, da seznam vsebuje sama pravilno opisana polja (nize v obliki e5, a8...), tako da je potrebno preveriti le, da seznam vsebuje vsa polja, da vsebuje vsako le enkrat (razen prvega, ki mora biti enako zadnjemu), in da so vse poteze pravilne, torej, da je konj vedno skočil tako, kot mora.

90. Družinsko drevo

V datoteki je shranjeno družinsko drevo: v vsaki vrstici sta po dve imeni, prvo ime je oče ali mati, drugo je ime otroka. Če je datoteka takšna,

```
bob mary
bob tom
bob judy
alice mary
alice tom
alice judy
renee rob
renee bob
sid rob
sid bob
```

```
tom ken
ken suzan
rob jim
```

prva vrstica pove, da ima Bob hči z imenom Mary, druga, da ima Bob tudi sina z imenom Tom... in zadnja pravi, da se je Robu rodil Jim. (Čestitamo!)

Napiši funkcijo, ki prebere datoteko v slovar, ki je za gornje podatke videti takole: `{'renee': ['rob', 'bob'], 'ken': ['suzan'], 'rob': ['jim'], 'sid': ['rob', 'bob'], ... , 'bob': ['mary', 'tom', 'judy']}`. Funkcija naj kot argument prejme ime datoteke, kot rezultat pa vrne slovar.

Nato napiši še funkciji, ki za podano osebo izpiše vse njene otroke in vse njene vnuke.

Bi znal napisati tudi funkcijo, ki za dano osebo izpiše njegove starše in prastarše?

91. Naključno generirano besedilo

Napisati želimo program, ki bo naključno generiral besedilo. Seveda ni dobro, da si samo naključno izbiramo besede in jih lepimo skupaj, saj bi tako dobili nekaj povsem neberljivega. Naloge se bomo lotili malo pametneje.

Recimo, da ima program na voljo nek tekst, iz katerega se lahko uči. Naš naključni tekst bomo začeli s poljubno besedo. Nadaljujemo tako, da se vprašamo, katere besede se v učnem tekstu pojavijo za to besedo; naključno izberemo eno od njih. Tretja beseda bo ena od besed, ki sledijo drugi… in tako naprej.

Naloge se bomo lotili po korakih.

Napiši funkcijo `nasledniki(s)`, ki vrne slovar, katerega ključi so besede, vrednosti pa seznami besed, ki sledijo tej besedi.

```
>>> nasledniki('in to in ono in to smo mi')
{'smo': ['mi'], 'to': ['in', 'smo'], 'ono': ['in'], 'in': ['to', 'ono', 'to']}
```

Za besedo `smo` se pojavi samo beseda `mi`, medtem ko se recimo za besedo `to` pojavita tako beseda `smo` kot tudi beseda `in`.

Nato napišite funkcijo `filozofiraj(besede, dolzina)`, ki prejme slovar, kakršen je gornji, in vrne naključno generirano besedilo podane dolžine.

92. Grde besede

Napiši program, ki v nizu nadomesti vse grde besede z naključno izbrano lepo sopomenko. Pri tem naj uporablja slovar, ki se začne, recimo, takole:

```
grde_besede = {
    'kreten': ['kljukec'],
    'idiot': ['mentalno zaostala oseba', 'omejen clovek']
}
```

Tako se niz "Joj ta Python spet se počutim kot idiot" pretvori v, recimo, "Joj ta Python spet se počutim kot mentalno zaostala oseba". Če ne obvladaš regularnih izrazov, predpostavi, da so v besedilu le besede, ločene s presledki, drugih ločil pa ni. Vse grde besede so pisane z malimi tiskanimi črkami.

93. Kronogrami

Veliko latinskih napisov, ki obeležujejo kak pomemben dogodek, je napisanih v obliki kronograma: če seštejemo vrednosti črk, ki predstavljajo tudi rimske številke (I=1, V=5, X=10, L=50, C=100, D=500, M=1000), dajo letnico dogodka. Tako, recimo, napis na cerkvi sv. Jakoba v Opatiji, CVIVS IN HOC RENOVATA LOCO PIA FVLGET IMAGO SIS CVSTOS POPVLI SANCTE IACOBE TVI, da vsoto 1793, ko je bila cerkev prenovljena (o čemer govori napis).

Pri tem obravnavamo vsak znak posebej: v besedil EXCELSIS bi prebrali X+C+L+I = 10+100+50+1 = 161 in ne XC + L + I = 90 + 50 + 1 = 141.

Napiši funkcijo, ki za podani napis vrne letnico, ki se skriva v njem.

94. Posebnež

Koliko bombonov ima kateri otrok, shranjujemo v slovarju, katerega ključ je ime otroka, pripadajoča vrednost pa število bombonov.

Recimo, da imajo vsi otroci enako bombonov, le eden jih ima več ali manj kot ostali. Napišite funkcijo `posebnez(bomboni)`, ki prejme takšen slovar (z vsaj tremi elementi) in vrne ime tega otroka.

95. Sumljive besede

Beseda je sumljiva, če ne poseduje črke, katero posedujejo vse druge besede te povedi. Prejšnji stavek ima eno besedo, katera ustreza temu. Tale stavek pa ima drugo takšno.

Torej: vse besede prvega stavka imajo črko e, le beseda "sumljiva" ga nima. Prav tako imajo v drugem stavku vse besede črko e, razen besede "ima"; v njem je torej sumljiv "ima". V tretjem stavku je sumljiva beseda "drugo", ki nima črke "a", ki jo imajo vse druge besede tega stavka.

Napiši funkcijo `sumljiva(s)`, ki kot argument prejme stavek in vrne sumljivo besedo. Pri reševanju

a) lahko predpostaviš, da funkcija dobi niz, v katerem so le besede, ločene s presledki, brez ločil;

b) tega ne predpostaviš.

Namig: moral boš odkriti, katera črka določa sumljivost. Če ima stavek n besed, je to črka, ki se pojavi v n-1 besedah. Ko odkriješ, za katero črko gre, pa bo preprosto.

96. Transakcije

V začetku je imela Ana štiri zlatnike, Berta 8 in CIlka 10, torej `[('Ana', 4), ('Berta', 8), ('Cilka', 10)]`. Nato je dala Cilka Ani 3 zlatnike; potem je dala Cilka Ani še 2; na koncu je dala Ana Berti 2, kar zapišemo `[('Cilka', 'Ana', 3), ('Cilka', 'Ana', 2), ('Ana', 'Berta', 2)]`. Kdo ima na koncu največ?

Napiši funkcijo `transakcije(zacetek, dajatve)`, ki dobi gornja seznama in vrne ime najbogatejše osebe po koncu transakcij. Če je na koncu več enako bogatih najbogatejših oseb, naj vrne poljubno izmed njih.

Namig: delo si boš poenostavil, če bo funkcija takoj pretvorila sezmam v primernejšo podatkovno strukturo.

```
>>> transakcije([('Ana', 4), ('Berta', 8), ('Cilka', 10)],
                [('Cilka', 'Ana', 3), ('Cilka', 'Ana', 2),
                 ('Ana', 'Berta', 2)])
Berta
```

97. Natakar

Ko je prišel natakar, so naročile:

- Ana je naročila torto,
- Berta je naročila krof,
- Cilka je naročila kavo,
- Ana je naročila še kavo,
- Berta je rekla, da ne bo krofa,
- Cilka je rekla, da ne bo torte (no, saj je niti ni naročila; to natakar mirno ignorira),
- Berta je naročila torto.

Vse skupaj zapišemo takole: `[("Ana", "torta"), ("Berta", "krof"), ("Cilka", "kava"), ("Ana", "kava"), ("Berta", "-krof"), ("Cilka", "-torta"), ("Berta", "torta")]`. Seznam torej vsebuje pare nizov (oseba, jed), pri čemer se jed včasih začne z "-", kar pomeni, da stranka prekliče naročilo te jedi oz. pijače.

Napiši funkcijo `narocila(s)`, ki prejme takšen seznam in vrne slovar, katerega ključi so imena strank, vrednost pri vsakem ključu pa je seznam vsega, kar mora stranka na koncu dobiti.

Namig: če uporabljaš `defaultdict`, dobiš iz njega seznam tako, da pokličeš funkcijo `dict`. Če je, na primer, `po_strankah` objekt tipa `defaultdict`, bo `dict(po_strankah)` običajen slovar.

```
>>> narocila([('Ana', 'torta'), ('Berta', 'krof'), ('Cilka', 'kava'),
              ('Ana', 'kava'), ('Berta', '-krof'), ('Cilka', '-torta'),
              ('Berta', 'torta')])
{'Cilka': ['kava'], 'Berta': ['torta'], 'Ana': ['torta', 'kava']}
>>>
>>> narocila([('Ana', 'torta'), ('Ana', '-torta')])
{'Ana': []}
>>>
>>> narocila([('Ana', '-torta')])
{'Ana': []}  # Tu sme funkcija vrniti tudi prazen slovar, {}
```

98. Tečaji

Univerza v Ljubljani je začela ponujati spletne tečaje. Kdo je že opravil kateri tečaj, zapisujejo v slovar, katerega ključi so imena oseb, pripadajoče vrednosti pa množice tečajev, ki jih je dotična oseba opravila. Če je takšnemu slovarju ime, recimo, `tecaji` in je `tecaji['Ana']` enako `{'Zgodovina gozdarstva', 'Sodobno roparstvo', 'Feminizem na Kočevskem'}`, to pomeni, da je Ana opravila naštete tečaje. Napisati boste morali več funkcij.

- `opravil(ime, tecaj, tecaji)` naj v slovar `tecaji` pribeleži, da je oseba opravila `tecaj`. (Če je v preteklosti že opravila ta tečaj, naj funkcija ne spremeni ničesar.)
- `najbolj_ucen(tecaji)` naj vrne ime osebe, ki je doslej opravila največ tečajev. Če je takšnih oseb več, naj vrne poljubno med njimi.
- `vsi_tecaji(tecaji)` naj vrne množico imen vseh tečajev, ki se pojavijo v slovarju.
- `neopravljeni(ime, tecaji)` naj vrne imena tečajev, ki jih je že kdo opravil, oseba ime pa še ne.

Predpostaviti smete, da je tecaji tipa `defaultdict(set)` in ne navaden `dict`.

99. Lego

Vsebino škatle Legovih kock opišemo s slovarjem: ključi predstavljajo tip kocke, vrednosti pa so število kock takšne oblike. Zapis `{"A": 2, "B": 1, "C": 3}` naj pomeni, da škatla vsebuje dve kocki vrste A, eno kocko vrste B in tri kocke vrste C.

Na podoben način lahko opišemo kocke, ki jih potrebujemo za izdelavo neke reči. `{"A": 4, "C": 3}` pomeni, da potrebujemo štiri kocke A in tri kocke C.

Opazimo lahko, da s kockami iz gornje škatle ne moremo narediti spodnje reči, ker imamo premalo kock vrste A (imamo dve, potrebovali bi štiri). Prav tako ne moremo narediti `{"A": 2, "D": 3}`, ker nimamo kock vrste D. Lahko pa bi naredili, recimo `{"A": 1, "C": 2}`.

Napiši funkcijo `vsi_deli(skatla, potrebno`, ki vrne `True`, če vsebina škatle, opisana s slovarjem `skatla`, zadošča, da naredimo reč, za katere potrebujemo vse, kar piše v slovarju potrebno.

Podobna funkcija `kaj_manjka(skatla, potrebno)` naj vrne slovar delov, ki manjkajo, da bi lahko iz škatle, katere vsebina je opisana v `skatla`, naredili reč, katere sestavni deli so opisani v potrebno.

Funkciji ne smeta spremeniti nobenega od slovarjev, ki jih dobita kot argument!

```
>>> vsi_deli ({"A": 2, "B": 1, "C": 3}, {"A": 3, "C": 2, "D": 2})
False
>>> kaj_manjka({"A": 2, "B": 1, "C": 3}, {"A": 3, "C": 2, "D": 2})
{"A": 1, "D": 2}
>>> vsi_deli({"A": 2, "B": 1, "C": 3}, {"A": 2, "C": 2})
True
>>> kaj_manjka({"A": 2, "B": 1, "C": 3}, {"A": 2, "C": 2})
{}
```

100. Slepa polja

V neki igri je mogoče z vsakega polja priti na določena druga polja. Možne poteze opišemo s slovarjem, ki kot ključe vsebuje imena polj, kot vrednosti pa množice polj, na katere je mogoče priti s posameznega polja. Tako

```
poti = {"a": {"b", "c"}, "b": {"a", "c", "e", "f"}, "c": {"d", "e"}, "e": {"a"}}
```

pomeni, da s polja `a` pridemo na `b` in `c`; s polja `b` na `a`, `c`, `e` in `f`; s polja `c` na `d` in `e`; s polja `e` na `a`. Nekatera polja so "slepa": v gornjem primeru sta to polji `d` in `f`, s katerih ni mogoče priti na nobeno drugo polje. Slepa polja lahko prepoznamo po tem, da se ne pojavljajo kot ključi (v slovarju ne bomo imeli nikoli praznih množic).

Napiši funkcijo `slepa_polja(s)`, ki dobi slovar poti in vrne množico slepih polj.

```
>>> slepa_polja(poti)
{"d", "f"}
>>> slepa_polja({"a": {"b", "c"}, "c": {"b"}})
{"b"}
>>> slepa_polja({"a": {"b", "c"}, "c": {"b"}, "b": {"c"}})
set()
```

101. Inventar

Neka trgovina shranjuje svojo zalogo v takšnem slovarju.

```
{'sir': 8, 'kruh': 15, 'makovka': 10, 'pasja radost': 2, 'pašteta': 10,
 'mortadela': 4, 'klobasa': 7}
```

Funkcija `zaloga(inventar, izdelek)` naj pove, koliko izdelka `izdelek` imamo na zalogi. Če je gornji slovar z inventarjem shranjen v slovarju `inv`, mora klic `zaloga(inv, "makovka")` vrniti `10`.

Funkcija `prodaj(inventar, izdelek, kolicina)`, naj zmanjša količino izdelka `izdelek` v inventarju `inventar` za `kolicina`. Klic `prodaj(inv, "makovka", 3)` (prodali smo tri makovke) mora *spremeniti* slovar `inv` tako, da zmanjša vrednost pri ključu `"makovka"` za 3. Novi slovar je tedaj takšen:

```
{'sir': 8, 'kruh': 15, 'makovka': 7, 'pasja radost': 2, 'pašteta': 10,
 'mortadela': 4, 'klobasa': 7}
```

Končno, napiši funkcijo `primanjkljaj(inventar, narocilo)`, ki prejme dva slovarja: prvi predstavlja trenutni inventar, drugi pa, kaj neka stranka naroča. Funkcija mora vrniti nov slovar, v katerem bo zapisano, koliko česa moramo še kupiti, da bomo lahko stranki dobavili vse, kar si želi. Če bi, recimo, stranka želela tri paštete, devet klobas in eno pivo, mora funkcija vrniti slovar `{"klobasa": 2, "pivo": 1}`. Dve klobasi zato, ker jih imamo sedem, stranka pa jih želi devet. Paštet ne bo potrebno naročati, saj jih imamo dovolj. Pivo pa potrebujemo, saj nimamo nobenega.

102. Nedostopna polja

V neki igri je mogoče z vsakega polja priti na določena druga polja. Možne poteze opišemo s slovarjem, ki kot ključe vsebuje imena polj, kot vrednosti pa množice polj, na katere je mogoče priti s posameznega polja. Tako

```
poti = {"a": {"c"},
        "b": {"a", "c", "f"},
        "c": {"d", "e"},
        "g": {"a"}}
```

pomeni, da s polja `a` pridemo na `c`; s polja `b` na `a`, `c` in `f`; s polja `c` na `d` in `e`; s polja `g` na `a`.

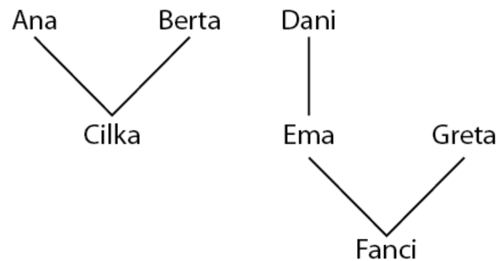
Poljem, na katera ne moremo priti z nobenega polja, bomo rekli nedostopna. V gornji igri sta to polji `b` in `g`.

Napiši funkcijo `nedostopna(s)`, ki prejme slovar možnih potez in vrne množico nedostopnih polj.

Namig: naloga sprašuje po poljih, ki se pojavijo med ključi, med vrednostmi pa ne.

103. Opravljivke

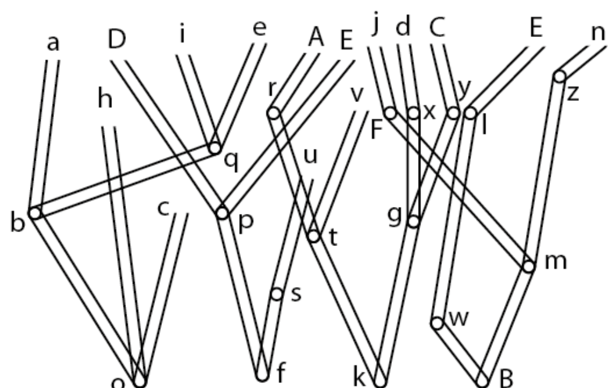
V tej in nekaj naslednjih nalogah se bomo ukvarjali z nekakšnimi mrežami. Za začetek si lahko predstavljamo, da gre za opravljivke, ki vsako novico sporočijo natančno eni osebi; Ana, na primer, vse izčveka Cilki, Berta prav tako pripoveduje Cilki, Dani Emi, Ema Fanči in Greta Fanči. Cilka in Fanči začuda vse obdržita zase.



Takšno mrežo bomo predstavili s slovarjem

```
čveke = {"Ana": "Cilka", "Berta": "Cilka", "Dani": "Ema", "Ema": "Fanci",
"Greta": "Fanci"}
```

Naredimo lahko tudi bolj zapletene mreže, na primer spodnji sistem cevi. V eno od cevi vržemo kroglico, zanima pa nas, kje prileti ven. Kroglico lahko vstavimo tudi na kateremkoli križišču ali celo na koncu (vstavimo jo na "o" in prileti iz "o").



Slovar zanj je takšen:

```
cevi = {"a": "b", "b": "o", "c": "o", "d": "x", "e": "q", "g": "k", "h": "o",
"i": "q", "j": "F", "l": "w",        "m": "B", "n": "z", "p": "f", "q": "b", "r":
"t", "s": "f", "t": "k", "u": "s", "v": "t", "x": "g", "y": "g", "z": "m", "A":
"r", "C": "y", "D": "p", "E": "l", "F": "m", "w": "B"}
```

Napiši naslednje funkcije.

- `kam(cevi, cev)`: prvi argument je slovar s sistemom, kot sta, recimo, gornja, drugi pa začetna cev. Če pokličemo, denimo, `kam(čveke, "Dani")` mora vrniti `"Fanči"` in če pokličemo `kam(cevi, "r")` mora vrniti `"k"`.
- `koliko_kam(cevi, zacetne)`: dobi slovar s sistemom in seznam začetnih pozicij ter vrne slovar, ki pove, kolikokrat smo končali na določenem mestu. Če pokličemo, recimo `koliko_kam(čveke, ["Ana", "Ana", "Greta", "Berta", "Cilka"])`, mora vrniti `{"Cilka": 4, "Fanci": 1}`.
- `najpogostejsa(cevi, zacetne)`: podobno kot prejšnja, le da vrne najpogostejši izhod. Če pokličemo `najpogostejsa(čveke, ["Ana", "Ana", "Greta", "Berta", "Cilka"])`, mora vrniti `"Fanci"`.

104. Vir in ponor

Nadaljujmo s podatki iz prejšnje naloge. Napiši funkciji `zacetne(cevi)` in `koncne(cevi)`.

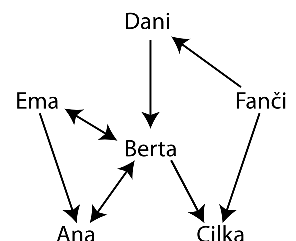
Prva naj vrne množico vseh cevi, ki nimajo vhodov, druga vse, ki so na koncu. Tako mora `zacetne(čveke)` vrniti `{"Ana", "Berta", "Dani", "Greta"}`, `koncne(čveke)` pa `{"Cilka", "Fanči"}`.

105. Viri

Napišite funkcijo `viri(cevi)`, ki vrne slovar, katerega ključi so končne cevi, vrednosti pa množice vseh točk, ki vodijo v to cev. V primeru opravljivk nas bosta ključa tisti dve, ki držita jezik za zobmi, pripadajoči vrednosti pa množici tistih, ki jih zalagata (posredno ali neposredno) z informacijami. Torej, `viri(čveke)` mora vrniti `{"Cilka": {"Ana", "Berta", "Cilka"}, "Fanci": {"Dani", "Ema", "Fanci", "Greta"}}`.

106. Sociogram

Skupino otrok smo prosili, da je vsak zapisal, kdo so njegovi prijatelji. Rezultat shranimo v slovar, recimo takšen: `sociogram = {"Ana": {"Berta"}, "Berta": {"Ana", "Cilka", "Ema"}, "Cilka": set(), "Dani": {"Berta"}, "Ema": {"Ana", "Berta"}, "Fanči": {"Cilka", "Dani"}}`.



Napiši funkciji

- `prijatelji(kdo, sociogram)`, ki za podano osebo in sociogram vrne *množico* tistih oseb, ki so podano osebo napisali za prijatelja, in
- `najbolj_priljubljen(sociogram)`, ki vrne tisto osebo v sociogramu, ki jo je največ drugih oseb navedlo za prijatelja.

107. Izplačilo

Napiši funkcijo `izplacilo(bankovci, znesek)`, ki dobi slovar, ki pove, koliko katerih bankovcev je v blagajni in znesek, ki ga je potrebno izplačati. Če je slovar enak `{100: 8, 20: 7, 10: 4}`, je v blagajni 8 stotakov, 7 dvajsetakov in 4 desetaki. Znesek je število (`int`), recimo 130. Znesek vedno izplačujemo tako, da začnemo z večjimi bankovci in nadaljujemo proti manjšim. Če je potrebno izplačati 130 evrov, bomo izdali en bankovec za 100, enega za 20 in enega za 10 in ne, recimo, šestih bankovcev za 20 in enega za 10 ali pa enega za 100 in treh za 10 – razen, kadar ne gre drugače, ker, recimo, nimamo bankovca za 100.

Funkcija ne sme vračati ničesar, temveč naj spremeni slovar `bankovci`, ki ga je dobila kot argument, tako, da bo odražal število bankovcev, ki ostanejo v blagajni po izplačilu. Če katere vrste bankovcev ni več, jih vrzite tudi iz slovarja: če je potrebno izplačati 800 evrov, mora biti vsebina slovarja po tem `{10: 4, 20: 7}` in ne `{100: 0, 20: 7, 10: 4}`. Če je potrebno izplačati 950 evrov, bo vsebina slovarja `{10: 3}`. Predpostavi smeš, da ima blagajna vedno dovolj bankovcev, da lahko opravi izplačilo.

108. Dopisovalci

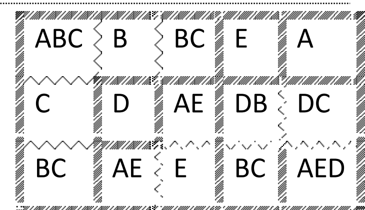
V tej nalogi bomo delali s slovarji, katerih ključi bodo imena ljudi, pripadajoče vrednosti pa množice imen vseh, ki jim je dotični pisal. Če je takšen slovar, recimo, `relacije` in je `relacije['Ana']` enako `{'Berta', 'Cilka', 'Dani'}`, to pomeni, da je Ana doslej pošiljala sporočila Berti, Cilki in Dani. Napiši funkcije

- `dopis(kdo, komu, relacije)` naj v slovar relacije pribeleži, da je kdo pisal komu. (Če je kdo v preteklosti že pisal komu, dopis ne bo spremenil ničesar.)
- `najzgovornejsi(relacije)` naj vrne ime osebe, ki je doslej pisala največ različnim osebam. Če je takšnih več, naj vrne poljubno med njimi.
- `vse_osebe(relacije)` naj vrne množico vseh oseb v slovarju – tako teh, ki so komu pisale, kot tistih, ki jim je kdaj kdo pisal.
- `neznanci(ime, relacije)` naj vrne množico imen vseh tistih oseb, ki so v slovarju, vendar jim oseba ime ni nikoli pisala.

Predpostaviti smeš, da je `relacije` tipa `defaultdict(set)` in ne navaden `dict`.

109. Zaporniki

Petnajst zapornikov se je znašlo v petnajstih celicah mednarodnega zapora. Dva zapornika se lahko pogovarjata, če sta v sosednjih celicah in govorita vsaj en skupni jezik (ali, učeno povedano, če je presek jezikov, ki jih govorita, neprazen ;).



Zapornik v prvi celici govori jezike A, B in C, zato se lahko pogovarja s svojim desnim sosedom (ki govori jezik B) in spodnjim sosedom (ki govori jezik C). Spodnji sosed se lahko pogovarja s še spodnejšim, ki govori B in C. Nesrečnik, ki čepi v zgornji desni celici govori jezik A in se ne more pogovarjati ne z levim sosedom (ta govori E) ne s spodnjim (ki govori DC). Pare, ki se lahko pogovarjajo, smo v gornjem primeru označili tako, da smo "stanjšali" zid med celicama.

Napiši funkcijo `sogovorniki(zapor)`, ki prejme razpored zapornikov v obliki `[["ABC", "B", "BC", "E", "A"], ["C", "D", "AE", "DB", "DC"], ["BC", "AE", "E", "BC", "AED"]]` in vrne število parov zapornikov, ki se bodo lahko pogovarjali. V gornjem primeru funkcija vrne 9.

Funkcija naj bo seveda splošna: sprejema naj tudi drugačne konfiguracije zaporov in več jezikov. Seveda pa bo en jezik vedno opisan z eno črko niza.

110. Ograje

Imamo naselje vrtičkarjev, ki je pravokotne oblike in je razdeljeno na *n*x*m* kvadratnih polj s stranico 1. Med dvema poljema je ograja, če imata različna lastnika. Primer je narisan na desni. Kdo je lastnik katerega vrtička, je zapisano v seznamu nizov; za primer na desni bi bil seznam takšen: `["AAABC", "ABCDC", "ACCDA"]`.

A	A	A	B	C
A	B	C	D	C
A	C	C	D	A

Napišite funkcijo `ograje(s)`, ki dobi seznam v tej obliki in vrne skupno dolžino vsej ograj – tako teh med vrtički kot vseh okrog njih.

Namig: morda bo lažje reševati tako, da najprej izračunamo, koliko ograje bi potrebovali, če bi bili vsi lastniki različni. Nato od tega odštejemo, koliko je meja, med katerimi je ograja odveč.

111. Trgovanje

Nekdo je za desko **ali** steklenico pripravljen dati pašteto. Nekdo drug je pripravljen za knjigo ali vilice dati svečo, desko ali papir. Nekdo tretji je za ribo ali pašteto pripravljen dati tipkovnico ali zaslon. To zapišemo kot

```
ponudbe = [({"deska", "steklenica"}, {"pašteta"}),
           ({"knjiga", "vilice"}, {"sveča", "deska", "papir"}),
           ({"riba", "pašteta"}, {"tipkovnica", "zaslon"})]
```

Če je tako, lahko zamenjamo vilice za desko, desko za pašteto in pašteto za zaslon **(**`["vilice", "deska", "pašteta", "zaslon"]`). Zamenjamo lahko tudi, recimo, desko za ribo. Ne moremo pa zamenjati vilic za papir, papirja za pašteto in paštete za tipkovnico (`["vilice", "papir", "pašteta", "tipkovnica"]`), ker na vsem božjem svetu ni nikogar, ki bi menjal papir za pašteto.

- Napiši funkcijo `obstaja(dam, dobim, ponudbe)`, ki pove (`True ali False`), ali v seznamu `ponudbe` obstaja kdo, ki bi za stvar `dam` dal stvar `dobim`. Klic `obstaja("pašteta", "zaslon", ponudbe)`, vrne `True`.
- Nato napiši funkcijo `menjave(zaporedje_stvari, ponudbe)`, ki pove, ali je možno podano zaporedje menjav. Klic `menjave(["deska", "pašteta", "zaslon"], ponudbe)` vrne `True`, `menjave(["vilice", "papir", "pašteta", "tipkovnica"], ponudbe)` pa `False`.

112. Ne na lihih

Napiši funkcijo `ne_na_lihih(s)`, ki prejme seznam, in vrne množico vseh elementov seznama, ki se nikoli ne pojavijo na lihih mestih (indeksih). Klic `ne_na_lihih([12, 17, 17, 5, 3])` vrne

{12, 3}. 5 je izpuščena, ker se pojavi samo na lihem mestu, 17 pa se sicer na sodem, vendar tudi na lihem.

Malo razmisli, preden se zaprogramiraš. Naloga je v resnici trivialna.

113. Sopomenke

Sopomenke lahko predstavimo z množico besed, kot, recimo `{"fant", "deček", "pob"}`. Takšne množice lahko zberemo v seznam, recimo `[{"fant", "deček", "pob"}, {"cesta", "pot", "kolovoz", "makadam"}, {"kis", "jesih"}]`.

Napišite funkcijo `predelaj(stavek, sopomenke)`, ki dobi stavek in sopomenke, ter vrne stavek, v katerem so vse besede, ki imajo sopomenke, zamenjane z naključno izbrano sopomenko. Tako lahko `predelaj("fant in dekle sta vzela pot pod noge", [{"fant", "deček", "pob"}, {"cesta", "pot", "kolovoz", "makadam"}, {"kis", "jesih"}, {"noge", "tace"}, {"dekle", "punca", "frajla"}])` vrne, recimo `"pob in punca sta vzela kolovoz pod tace"`.

114. Stavka z istim pomenom

Sopomenke lahko predstavimo z množico besed, kot, recimo `{"fant", "deček", "pob"}`. Takšne množice lahko zberemo v seznam, recimo `[{"fant", "deček", "pob"}, {"cesta", "pot", "kolovoz", "makadam"}, {"kis", "jesih"}]`.

Napišite funkcijo `sopomena(stavek1, stavek2, sopomenke)`, ki pove, ali `stavek1` in `stavek2` pomenita isto. Tako mora, recimo `sopomena("fant in dekle sta vzela pot pod noge", "pob in punca sta vzela kolovoz pod tace", [{"fant", "deček", "pob"}, {"cesta", "pot", "kolovoz", "makadam"},{"kis", "jesih"}, {"dekle", "punca"}, {"noge", "tace"}])` vrniti `True`. Predpostavite lahko, da v stavku ni ločil. Ne vznemirjajte se zaradi sklonov.

115. Združi - razmeči

Napiši funkcijo `zdruzi(s)`, ki prejme seznam s ter vrne slovar, katerega ključi so elementi s-a, pripadajoče vrednosti pa množice indeksov, kjer se ta element pojavlja. Klic `zdruzi([3, 1, 12, 3, 7, 12])` mora vrniti `{1: {1}, 3: {0, 3}, 7: {4}, 12: {2, 5}}`.

Napiši še funkcijo `razmeci(s)`, ki dela ravno obratno – prejme takšen slovar in vrne seznam.

116. Podajanje daril

Nekateri ljudje podarijo čokolade naprej, drugi jih pojejo. Če imamo seznam `[(8, 2), (1, 8), (5, 1), (4, 42)]`, to pomeni, da oseba 8 daje čokolade osebi 2, oseba 1 osebi 8 in tako naprej.

Napiši funkcijo `dolzina_poti(s, o)`, ki prejme tak seznam in številko osebe, ki ji damo čokolado, ter vrne število podaj, preden bo čokolada pojedena.

Klic `dolzina_poti([(8, 2), (1, 8), (5, 1), (4, 42)], 5)` vrne `3`. Če namreč damo čokolado osebi `5`, jo bo ta dala osebi `1`, oseba `1` osebi `8` in oseba `8` osebi `2`, ki jo poje. Torej tri podaje.

Seznam lahko tudi zelo dolg; morda se ga splača znotraj funkcije pretvoriti v kaj prikladnejšega za to nalogo.

117. Požrešneži

Nekateri ljudje podarijo čokolade naprej, drugi jih pojejo. Če imamo seznam `[(3, 1), (8, 2), (1, 8), (4, 5)]`, to pomeni, da oseba 3 daje čokolade osebi 1, oseba 8 osebi 2 in tako naprej.

Napiši funkcijo `ne_daje_naprej(s)`, ki prejme takšen seznam in vrne množico številk oseb, ki čokolad ne dajejo naprej. V gornjem primeru je to `{2, 5}`.

Upoštevaj, da je seznam lahko tudi zelo dolg; morda se ti splača znotraj funkcije uporabiti kaj prikladnejšega za to nalogo.

118. Ne maram

Imamo seznam parov oseb, ki nočejo biti skupaj, na primer `[("Ana", "Cilka"), ("Berta", "Ana"), ("Berta", "Dani")]`. Imamo vrstni red oseb, na primer, `["Ana", "Berta", "Cilka", "Dani", "Ema"]`. Potrebujemo samo še funkcijo, `preveri_vrsto(vrsta, prepovedani)`, ki vrne `True`, če bodo uvrščenke zadovoljne z vrsto in `False`, če ne. Za gornji primer vrne `False`, ker Ana in Berta nočeta biti ena zraven druge.

Funkcija naj bo napisana tako, da deluje hitro tudi, če je oseb zelo veliko.

119. Najmanjši unikat

Napiši funkcijo `najmanjsi_unikat(s)`, ki prejme seznam nekih stvari (lahko so števila, nizi, … karkoli, kar je možno primerjati) in vrne najmanjši element, ki se v seznamu pojavi le enkrat. Če takega ni, ker je seznam prazen ali pa se vsi pojavijo večkrat, ne vrne ničesar (torej `None`).

120. Bingo

Igra Bingo poteka tako, da ima vsak igralec listek z nekaj številkami. Voditelj žreba številke. Če se izžrebana številka nahaja na igralčevem listku, jo ta prečrta. Zmaga igralec, ki prvi prečrta vse številke.

Vedeževalka nam je napovedala vrstni red, v katerem bodo žrebane številke. Imamo seznam listkov za Bingo; izbrali bi radi tistega, na katerem bodo prej prečrtane se številke.

Napiši funkcijo `bingo(listki, vrstni_red)`, ki prejme listke (seznam seznamov števil) in vrne listek, na katerem bodo najprej prečrtane vse številke. Klic

```
bingo([[4, 1, 2, 3, 5], [6, 1, 2, 3, 4], [7, 6, 4, 3, 2]],
      [4, 2, 8, 3, 1, 6, 5, 7])
```

vrne [6, 1, 2, 3, 4] (saj bo ta očitno prehitel prvega in zadnjega, zaradi 5 in 7).

Funkcija naj ne spreminja podanega seznama!

121. Trki besed

Podana je funkcija skrij(beseda), ki na določen način predeluje besede: kot argument dobi besedo (npr. "HANA") in kot rezultat vrne predelano besedo ("A6N3H1"). Kako funkcija deluje, ni pomembno. Še več, testi lahko uporabljajo različne funkcije za predelavo besed.

Napiši funkcijo trk(besede), ki dobi seznam besed in vrne par besed, ki ju funkcija skrij predela v isto besedo. Če je takšnih parov več, lahko vrne poljubnega med njimi. Če se noben par ne spremeni v isto besedo, naj vrne None.

Rekurzija

Prve naloge iz rekurzije bodo na temo celjskih grofov: predpostavljale bodo, da je njihova rodbina shranjena v globalni spremenljivki `rodovnik`. Rodovnik je shranjen kot slovar, katerega ključi so imena, vrednosti pa seznami imen potomcev. Ti so prazni, če je dotična oseba brez potomcev. (Rodovnik je podoben resničnemu, a poenostavljen za potrebe naloge.)

```
rodovnik = {'Ulrik I.': ['Viljem'], 'Margareta': [],
            'Herman I.': ['Herman II.', 'Hans'], 'Elizabeta II.': [],
            'Viljem': ['Ana Poljska'], 'Elizabeta I.': [], 'Ana Poljska': [],
            'Herman III.': ['Margareta'], 'Ana Ortenburška': [],
            'Barbara': [], 'Herman IV.': [], 'Katarina': [], 'Friderik III.': [],
            'Herman II.': ['Ludvik', 'Friderik II.', 'Herman III.',
                           'Elizabeta I.', 'Barbara'],
            'Ulrik II.': ['Herman IV.', 'Jurij', 'Elizabeta II.'],
            'Hans': [], 'Ludvik': [], 'Jurij': [],
            'Friderik I.': ['Ulrik I.', 'Katarina', 'Herman I.',
                            'Ana Ortenburška'],
            'Friderik II.': ['Friderik III.', 'Ulrik II.'] }
```

Kadar naloge govorijo o potomstvu neke osebe, mislijo na njegove otroke, vnuke, pravnuke in tako naprej, kolikor daleč gre. Kadar govorijo o rodbini, imajo v mislih potomstvo in še to osebo samo.

122. Preštej vnuke

Napiši funkcijo `prestej_vnuke(oseba)`, ki vrne število vnukov podane osebe.

123. Poišči rojaka

Napiši funkcijo `poisci_rojaka(oseba, ime)`, ki pove, ali je komu iz rodbine osebe `oseba` ime `ime`.

124. Poišči potomca

Napiši funkcijo `poisci_potomca(oseba, ime)`, ki pove, ali je kateremu od potomcev osebe `oseba` ime `ime`.

125. Preštej rodbino

Nekdo povabi na kosilo vso svojo rodbino – sebe, svoje sinove, vnuke, pravnuke in tako naprej. Koliko krožnikov za juho potrebuje? Pomagaj mu poiskati odgovor tako, da napišeš funkcijo `prestej_rodbino(oseba)`.

126. Preštej potomce

Zdaj pa stori isto, vendar štej le potomce, brez osebe; funkciji bo ime `prestej_potomce`.

127. Najdaljše ime v rodbini

Napiši funkcijo `najdaljse_ime(oseba)`, ki vrne najdaljše ime, ki se pojavi v rodbini določene osebe.

128. Globina rodbine

Napiši funkcijo `globina(oseba)`, ki vrne globino rodbine določene osebe. Če oseba nima otrok, je globina 1. Če ima kakega otroka, vendar nihče od njih nima otrok, je globina 2 ... in tako naprej.

129. Kolikokrat ime

Napiši funkcijo `kolikokrat_ime(oseba, ime)`, ki pove, kolikokrat se v rodbini osebe pojavi določeno `ime`. Pri tem `ime` nima dodatnih oznak (številk ipd.). V rodbini Friderika I. so, recimo, trije Frideriki (različno oštevilčeni), dve Ani, ena Barbara in nič Francljev.

130. Koliko žensk

Napiši funkcijo `zensk_v_rodbini(oseba)`, ki vrne število žensk v rodbini osebe `oseba`. Prepodstavi, da je oseba ženska, če se njeno ime (to je, prvi del, pred raznimi številkami in krajevnimi nadimki) konča s črko "a".

131. Naštej rodbino

Napiši funkcijo `vsa_rodbina(oseba)`, ki vrne množico (kot `set`) z vso rodbino podane osebe.

132. Naštej potomce

Napiši funkcijo `vse_potomstvo(oseba)`, ki vrne množico (kot `set`) potomcev podane osebe.

133. Največ otrok

Napiši funkcijo `najvec_otrok(oseba)`, ki pove, kolikšno je največje število otrok pri kateremkoli članu rodbine določene osebe.

134. Največ vnukov

Napiši funkcijo `najvec_vnukov(oseba)`, ki pove, koliko vnukov ima oseba z največ vnuki.

Nasvet: mogoče vam bo lažje, če najprej napišete funkcijo `vnukov(oseba)`, ki pove, koliko vnukov ima `oseba`.

135. Največ sester

Napiši funkcijo `najvec_sester(oseba)`, ki pove, koliko sester ima oseba z največ sestrami. Predpostavili bomo, da je ime žensko, če se konča s črko "a".

Pazi: v družini z otroki Ana, Berta, Jože sta dve hčeri in nekdo (Jože) ima dve sestri. V družini z otrokoma Ano in Berto sta prav tako dve hčeri, vendar ima vsako od njiju le eno sestro.

Nasvet: mogoče bo lažje, če najprej napišeš funkcijo `sester_pod(ime, rodovnik)`, ki pove, koliko sester ima tisti otrok osebe `ime`, ki ima največ sester.

136. Najplodovitejši

Spremeni funkcijo tako, da ne bo povedala, koliko je največje število otrok v kaki družini, temveč kdo je njihov cenjeni oče. Funkciji naj bo ime `najvec_otrok_kdo(oseba)`. Poskusi jo napisati tako, da bo vsaka oseba še vedno "zastavljala vprašanja" samo svojim otrokom; nihče ne sme spraševati po številu otrok, ki jih ima neka druga oseba.

137. Brez potomca

Napiši funkcijo `brez_potomca(oseba)`, ki vrne poljubno osebo iz rodbine podane osebe, ki nima nobenega otroka.

Razmisli o rekurzivni in nerekurzivni obliki funkcije.

138. Vsi brez potomca

Dopolni funkcijo `brez_potomca` iz prejšnje naloge. Imenujmo jo `brez_potomcev(oseba)`, vrniti pa mora množico imen vseh oseb iz rodbine podane osebe, ki nimajo potomcev.

Namig: Če dotična oseba nima otrok, vrne množico z lastnim imenom. Sicer kliče otroke in vrne unijo množic, ki jo dobi od otrok.

139. Kako daleč

Napiši funkcijo `globina_do(oseba, ime)`, ki pove, kako globoko v rodbini osebe `oseba` je oseba z imenom `ime`. Če, recimo, vprašamo Friderika I., kako daleč ima do Hansa, mora odgovoriti 2, saj je Hans njegov vnuk. Če ga vprašamo, kako daleč ima do Friderika I., mora odgovoriti 0 (saj je to on sam). Če ga vprašamo, kako daleč ima do Franclja III., naj odgovori -1, s čimer bo povedal, da v njegovi rodbini vendar ni nobenih Francljev.

140. Pot do

Reši podobno nalogo, kot je prejšnja, le da naj funkcija - poimenujmo jo `pot_do(oseba, ime)` – ne vrne razdalje do določene osebe, temveč pot do nje. Če Friderika II. vprašamo, kakšna je njegova pot do Hansa, mora odgovoriti `["Friderik I.", "Herman I.", "Hans"]`. Če ga

vprašamo po poti do Friderika I. (torej njega samega), bo rekel samo `["Friderik I."]`. Če ga vprašamo po poti do Franclja, naj vrne `None`, ker ni nikjer nobenih Francljev.

141. Zaporedja soimenjakov

Tale naloga je pa malo hujši izziv: napiši funkcijo `najdaljse_zaporedje(oseba)`, ki vrne najdaljše zaporedje enakih imen v rodbini osebe `oseba`. Klic `najdaljse_zaporedje("Friderik I.")` mora vrniti `(3, "Herman")`, kar pomeni, da so v rodbini trije zaporedni Hermani (eden od sinov Hermana I. je Herman II. in eden od njegovih sinov je Herman III.). Pri tem ne glej številk, le imena (dovoljeno bi bilo tudi zaporedje Herman Strašni, Herman Strašnejši, Herman Najstrašnejši). Klic `najdaljse_zaporedje("Friderik II.")` vrne `(2, "Friderik")`. Klic `najdaljse_zaporedje("Hans")` vrne `(1, "Hans")`.

142. Fakulteta

Fakulteto, *n*!, navadno računamo kot produkt n števil; 5! = 1*2*3*4*5. Lahko pa bi jo definirali tudi takole: $n! = n\,(n-1)!$ Če jo lahko tako definiramo - pa jo tako še sprogramirajmo!

Da se stvar izteče se moramo dogovoriti še, da je 0! enako 1.

143. Fibonacijeva števila

Napiši rekurzivno funkcijo `fibonaci(n)`, ki izračuna n-to Fibonacijevo število F_n. Fibonacijeva števila so definirana s $F_n = F_{n-1} + F_{n-2}$, pri čemer je $F_0 = F_1 = 1$. Manj matematično: *n*-to Fibonacijevo število je vsota prejšnjih dveh, ničto in prvo pa sta 1.

144. Vsota seznama

Napiši rekurzivno funkcijo `vsota(s)`, ki izračuna vsoto elementov seznama `s`, ki vsebuje sama števila. Vsota elementov praznega seznama je 0. Vsota nepraznega seznama je enaka vsoti prvega elementa in vsoti preostalega seznama (`s[1:]`).

145. Iskanje elementa

Napiši rekurzivno funkcijo `vsebuje(s, x)`, ki pove, ali seznam `s` vsebuje element `x`. Seznam vsebuje element `x`, če `s` ni prazen in je bodisi prvi element `s` enak `x` bodisi ostanek seznama vsebuje `x`.

146. Enaka seznama

Napiši rekurzivno funkcijo `enaka(s, t)`, ki pove (s `True` ali `False`) ali sta dva seznama enaka ali ne. Dva seznama sta enaka, če sta oba prazna ali pa sta oba neprazna in imata enaka prva elementa in sta enaka tudi ostanka seznama (vse razen prvega elementa).

147. Palindrom

Napiši rekurzivno funkcijo `palindrom(s)`, ki preveri, ali je podani niz `s` palindrom.

Niz je palindrom, če se naprej bere enako kot nazaj. Povedano rekurzivno: niz je palindrom, če je krajši od dveh znakov ali pa je prvi znak enak zadnjemu in je niz med drugim in predzadnjim znakom palindrom.

148. Preverjanje Fibonacija

Zaporedje je Fibonacijevo, če je vsak člen enak vsoti prejšnjih dveh.

Seznam vsebuje Fibonacijevo zaporedje, če ima manj kot tri elemente ALI pa je zadnji element vsota predzadnjih dveh in je tudi seznam brez zadnjega elementa Fibonacijev.

Napiši ***rekurzivno*** funkcijo `je_fibo(s)`, ki preveri, ali podani seznam vsebuje Fibonacijevo zaporedje.

149. Naraščajoči seznam

Seznam je naraščajoč, če je prvi element manjši od drugega in če je naraščajoč tudi seznam od drugega elementa naprej. Poleg tega so seveda naraščajoči vsi seznami z manj kot dvema elementoma.

Napišite rekurzivno funkcijo `narascajoci(s)`, ki vrne `True`, če je podani seznam naraščajoč, in `False`, če ni.

150. Vsota gnezdenega seznama

Napiši funkcijo `vsota2(s)`, ki sešteje elemente gnezdenega seznama (seznama, ki vsebuje tudi podseznamе in ti svoje podseznamе...). Pomagaj si s funkcijo `isinstance(x, tip)`, ki pove, ali je podani x tipa `tip`.

Namig: vsota elementov seznama je enaka vsoti vsot podseznamov + vsota elementov, ki niso podseznami. Se pravi, pojdi čez elemente seznama (npr. `for e in xs`). Če vidiš, da gre za seznam (`isinstance(e, list)`), ga vprašaj, kakšna je njegova vsota (vključno z morebitnimi podseznami v njem); to prištej k skupni vsoti, ki jo računaš. Če pa ne gre za seznam, je to številka in jo preprosto prišteješ.

151. Obrni

Napiši funkcijo `obrni(s)`, ki prejme niz in vrne isto besedilo zapisano z desne proti levi. Tako mora, recimo, `obrni("janez demšar")` vrniti `"rašmed zenaj"`.

Obrnjen niz dobimo tako, da k obrnjenemu nizu brez prve črke prištejemo prvo črko. Torej, če hočeš obrniti "demšar", moraš obrniti "emšar" in k temu prišteti "d".

Funkcija mora biti rekurzivna. Rešitev `def obrni(s): return s[::-1]` ni pravilna.

152. Zrcalo

Napiši funkcijo `zrcalo(s)`, ki dobi niz in vrne niz, ki vsebuje začetni niz, znak `|` in potem še enkrat začetni niz, vendar prezrcaljen. Tako naj `zrcalo("janez")` vrne `"janez|zenaj"`.

Nasvet: Najboljše, da najprej poskrbiš za prazen niz. Sicer pa je rešitev skoraj enaka kot rešitev ogrevalne naloga, le da prva črka ni samo na koncu, temveč tudi na začetku.

153. Sodo – lihi – rekurzivno

Sodo-lihi seznami so seznami, v katerih se izmenjujejo soda in liha števila; začeti se morajo s sodim številom. Liho-sodi seznami so zelo podobna reč, le da se morajo začeti z lihim številom.

Nekoliko rekurzivno povedano: seznam je sodo-lih, če se začne s sodim številom, ostanek pa je liho-sodi seznam. In obratno. Prazni seznami so sodo-lihi in liho-sodi.

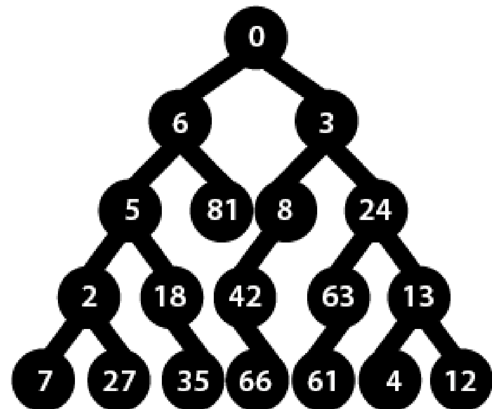
Napiši rekurzivno funkcijo `sodi_lihi(s)`, ki vrne `True`, če je seznam sodo-lih in `False`, če ni.

Nasvet: napiši tudi `lihi_sodi(s)`, saj ti bo v pomoč.

154. Kam?

Pod Meglenim gorovjem je, kot nekateri vedo že dolgo, nekateri pa so morali čakati na film, cel labirint votlin. Bilbu bi bilo veliko lažje, če bi imel zemljevid. Mi ga imamo.

Zemljevid lahko shranimo tudi v obliki slovarja, v katerem so ključi številke sob, vrednosti pa sobe, v katere pridemo po levi in po desni poti. Kadar kakšne poti ni, je ustrezni element enak `None`.




```
zemljevid = {0: [6, 3], 6: [5, 81], 3: [8, 24], 5: [2, 18], 81: [None, None],
8: [42, None], 24: [63, 13], 2: [7, 27], 18: [None, 35], 42: [None, 66],
63: [61, None], 13: [4, 12], 7: [None, None], 27: [None, None], 35: [None, None],
66: [None, None], 61: [None, None], 4: [None, None], 12: [None, None]}
```

Napiši funkcijo `prehodi_pot(zemljevid, soba, pot)`, ki kot argument prejme zemljevid (npr. zgornji slovar), začetno sobo (ta bo vedno 0, a to naj te ne moti) in pot, podano kot zaporedje znakov L in D (levo in desno). Kot rezultat mora vrniti sobo, v katero pelje podana pot.

```
>>> prehodi_pot(zemljevid, 0, "LLD")
18
>>> prehodi_pot(zemljevid, 0, "DDDD")
12
>>> prehodi_pot(zemljevid, 0, "L")
6
>>> prehodi_pot(zemljevid, 0, "")
0
```

Predpostaviti smeš, da je pot pravilna in se nikoli ne odpravi v hodnik, ki ga ni.

155. Razdalja do cilja

Nadaljujmo z meglenim gorovjem. Prstan se seveda *vedno* nahaja v sobi številka 42. Napiši funkcijo `globina_prstana(zemljevid, soba)`, ki prejme zemljevid in številko začetne sobe, kot rezultat pa vrne globino te sobe - torej dolžino poti do nje.

Na gornjem zemljevidu je globina sobe 42 enaka 3. Funkcija mora seveda delovati tudi na poljudnih drugih zemljevidih.

156. Pot do cilja

Napiši funkcijo `pot_do_prstana(zemljevid, soba)`, ki vrne pot do sobe številka 42. V gornjem primeru mora vrniti niz `"DLL"`.

157. Naprej nazaj

Žaba skače en korak daleč v eno od štirih smeri neba. Njeno pot opišemo z nizom, kot je `"SSVJVZSZJJ"`.

Če pozorno pogledaš prav tale niz, vidiš, da je šla žaba najprej po poti SSVJV, nato pa po isti poti nazaj: ZSZJJ. Napiši **rekurzivno** funkcijo `naprej_nazaj(s)`, ki vrne `True`, če niz opisuje pot, ki gre nekam in po tem po isti poti nazaj in `False`, če pot nima te lastnosti.

V pomoč: pot je takšna, če je dolga nič **ali** pa je zadnji element ravno obrat prvega (recimo prvi S in zadnji J ali prvi J in zadnji Z ali prvi V in zadnji Z ali prvi Z in zadnji V) **in** je takšen tudi niz med drugo in predzadnjo črko.

Lahko si pomagaš tudi s slovarjem `obratno = {"S": "J", "J": "S", "V": "Z", "Z": "V"}`. Ni pa treba.

158. Aritmetično zaporedje

Aritmetično zaporedje je zaporedje, v katerem je razlika med zaporednimi členi enaka, na primer 3, 7, 11, 15, 19, 23 ali 5, 3, 1, -1, -3, -5.

Z drugimi besedami, zaporedje je aritmetično, če je razlika med prvima dvema členoma enaka razliki med drugim in tretjim, poleg tega pa je aritmetično tudi zaporedje od drugega člena naprej. Vsako zaporedje, ki ima manj kot tri člene je, očitno, aritmetično.

Napiši *rekurzivno* funkcijo `aritmeticno(s)`, ki prejme zaporedje števil in vrne `True`, če je aritmetično in `False`, če ni.

159. Binarno

Napiši **rekurzivno** funkcijo `binarno(n)`, ki kot argument prejme število `n` in kot rezultat vrne niz s tem številom v dvojiškem zapisu.

```
>>> binarno(42)
'101010'
```

Namig: desno števko dobiš tako, da izračunaš ostanek po deljenju z 2. Pred njo pa moraš postaviti niz, ki ga dobiš, če v dvojiški zapis spremeniš število n, ki ga celoštevilsko deliš z 2.

160. Decimalno

Napiši **rekurzivno** funkcijo `decimalno(s)`, ki kot argument prejme niz s, ki predstavlja število v dvojiškem zapisu, kot rezultat pa naj vrne to število. Klic `decimalno('101010')` vrne `42`.

Namig: k zadnji števki prištej dvakratnik števila, ki ga dobiš, če vse razen zadnje števke pretvoriš iz dvojiškega zapisa v število.

161. Zadnje liho

Napiši *rekurzivno* funkcijo `zadnje_liho(s)`, ki vrne zadnje liho število v podanem seznamu ali `None`, če v njem ni lihih števil.

162. Indeksi

Napiši *rekurzivno* funkcijo `indeksi_rec(s, e)`, ki prejme seznam in vrednost ter vrne seznam indeksov, na katerih se pojavi ta vrednost. Klic `indeksi_rec([6, 2, 4, 6, 6, 3], 6)` vrne `[0, 3, 4]`.

163. Brez jecljanja

Napiši *rekurzivno* funkcijo `brez_jecljanja_rec(s)`, ki prejme seznam `s` in vrne nov seznam, ki ne vsebuje zaporednih ponovitev istega elementa. Klic `brez_jecljanja([1, 6, 3, 3, 1, 1, 1, 1, 2, 3, 5, 5, 1])` naj vrne `[1, 6, 3, 1, 2, 3, 5, 1]`.

164. Rekurzivni Collatz

Kako je videti Collatzovo zaporedje, smo že nekajkrat videli. Vzamemo poljubno število in z njim počnimo tole: če je sodo, ga delimo z 2, če je liho, pa ga pomnožimo s 3 in prištejmo 1. To ponavljamo, dokler ne dobimo 1.

Napiši *rekurzivno* funkcijo `collatz(n)`, ki pove, kako dolgo je Collatzovo zaporedje, ki se začne s številom `n`. Tako mora `collatz(42)` vrniti `9`, saj ima zaporedje, ki se začne z 42 devet členov: 42, 21, 64, 32, 16, 8, 4, 2, 1.

165. Sprazni

Napiši rekurzivno funkcijo `sprazni(s)`, ki prejme seznam, katerega elementi so `None` in seznami, katerih elementi so `None` in seznami in tako naprej. Funkcija naj vrne prav tak seznam,

a brez `None`ov. Če pokličemo `sprazni([None, None, [None], [None, None, []], None])`, dobimo `[[], [[]]]`.

Pomoč: funkcija sestavlja seznam. Če vidi `None`, ga preskoči. Če vidi seznam, ga doda v nastajajoči seznam ... a spraznjenega.

166. Rekurzivni štumfi, zokni, kalcete, kucjte

Ker smo moški, vemo, barvno slepi, obenem pa v družini z ženo in *n* otroki ni mogoče, da bi imel vsak same enake nogavice, se lahko znajdemo tako, da na vsako nogavico prišijemo številko. Enake nogavice imajo enako številko; ker imamo več enakih nogavic, ima lahko več nogavic enako številko.

Recimo, da iz pralnega stroja potegnemo nogavice s številkami `[1, 2, 3, 2, 3, 1, 3, 1, 1, 1, 1]`. Imamo torej tri pare nogavic 1, en par nogavic 2 in en par nogavic 3 in (eh, spet!) še eno 3 brez para.

Napiši rekurzivno funkcijo `brez_para(nogavica, nogavice)`, ki prejme številko nogavice in podoben seznam kot prva naloga. Vrniti mora `True`, če je nogavica brez para, in `False`, če ni.

Klic `brez_para(39, [41, 39, 39, 41, 41, 39, 39])` vrne `False` in klic `brez_para(41, [41, 39, 39, 41, 41, 39, 39])` vrne `True`, saj imamo eno 41 brez para.

Splošne vaje iz programiranja

167. Stopnice

Stopnišča z neenakimi višinami stopnic lahko opišemo tako, da navedemo višine stopnic, merjene od tal (ne od prejšnje stopnice). Po stopnicah, opisanih s takšnim seznamom, pleza robot, ki lahko stopi največ 20 cm visoko. Napiši funkcijo `kako_visoko(stopnice)`, ki kot argument prejme višine stopnic, rezultat, ki ga vrne, pa pove, kako visoko bo robot priplezal.

```
>>> kako_visoko([10, 20, 25, 45, 50, 71, 98, 110])
50
>>> kako_visoko([30, 40, 50])
0
>>> kako_visoko([10, 20, 30, 40, 45])
45
```

V prvem primeru robot pripleza do višine 50, nato pa se ustavi, ker je naslednja stopnica 21 cm višja od te, na kateri stoji. V drugem se ne more povzpeti niti na prvo stopnico. V zadnjem pripleza do vrha.

168. Drugi največji element

Napiši program, ki vrne drugi največji element v seznamu. Če se največji element pojavi večkrat, njegove ponovitve ne štejejo za drugi največji element: če imamo seznam `[5, 1, 4, 8, 2, 3, 8]`, je drugi največji element `5`, ne `8`.

169. Collatz 2

Spomni se naloge Collatzova domneva. Nekoliko težja naloga je napisati program, ki izpiše tisto število med 1 in 100000, ki vodi v najdaljše zaporedje, in dolžino tega zaporedja (vključno s prvim številom in enico). (Da boste vedeli, ali ste naračunali pravi rezultat: število, ki da najdaljše zaporedje, je deljivo le z dvema različnima prašteviloma - z enim od njiju kar petkrat, z drugim pa le enkrat.

170. Delnice

Imamo seznam z gibanji vrednosti določene delnice v posameznih mesecih 2009. Seznam je lahko, na primer, takšen:

```
delnica = [1, -2, -4, 1, 2, -1, 3, 4, -2, 1, -5, -5]
```

Delnica na gornjem seznamu je pridobila eno točko januarja, februarja je izgubila dve, marca je izgubila štiri, aprila pridobila eno... Napišite funkcijo `posrednik(s)`, ki dobi seznam, kakršen je gornji in vrne par (terko), ki pove, kdaj je bilo delnico pametno kupiti in prodati, da bi z njo dosegli največji dobiček. Delnico bomo kupili in prodali samo enkrat.

```
>>> delnica = [1, -2, -4, 1, 2, -1, 3, 4, -2, 1, -5, -5]
>>> kupim, prodam = posrednik(delnica)
>>> kupim
3
>>> prodam
8
```

Delnico je smiselno kupiti v tretjem mesecu (če je januar "ničti ") in prodati pred osmim. Vmes vrednost delnice sicer enkrat tudi nekoliko pade (-1), vendar nas to ne moti, saj po tem še pošteno zraste. Skupaj smo v tem času zaslužili 1 + 2 – 1 + 3 + 4 = 9 točk.

171. Spremembe smeri

Napiši funkcijo `spremenb_smeri(s)`, ki kot argument prejme seznam z vrednostmi delnice ob koncu dnevnega trgovanja, kot rezultat pa vrne število sprememb gibanja. Če delnica nekaj dni narašča, nato pada, nato narašča, je smer spremenila dvakrat (iz naraščanja v padanje in iz padanje v naraščanje).

Pri, recimo, seznamu `[1, 2, 3, 0, -1]` vrne `1`, saj vrednost delnice najprej narašča, nato pada. Pri `[-1, -2, 0, 1, -10, -5, 8]` vrne `3` – pada, narašča, pada, narašča. Pri `[1, 2, 3, 4]` vrne `0`, saj vrednost ves čas narašča.

Predpostaviti smeš, da se nikoli ne zgodi, da bi imela delnica dva dni zapored isto vrednost.

172. Sekajoči se krogi

Napiši funkcijo `sekajo(krogi)`, ki dobi seznam krogov, podanih s terkami (x, y, r), kjer sta x in y koordinati središča kroga, r pa polmer. Funkcija naj vrne `True`, če se vsaj dva kroga sekata in `False`, če se ne. Ker gre za kroge (in ne krožnice), se dva kroga sekata tudi, če je eden od njiju v celoti znotraj drugega.

173. Največ n-krat

Napiši funkcijo `najvec_n(s, n)`, ki kot argument prejme seznam `s` in največje dovoljeno število ponovitev elementov na seznamu, `n`. Funkcija naj vrne nov seznam, v katerem se vsak element pojavi največ `n`-krat, vse nadaljnje ponovitve pa se pobrišejo.

Če ti kaj pomaga, smeš predpostaviti, da seznami niso dolgi.

```
>>> najvec_n([1, 2, 3, 1, 1, 2, 1, 2, 3, 3, 2, 4, 5, 3, 1], 3)
[1, 2, 3, 1, 1, 2, 2, 3, 3, 4, 5]
```

Naloge se loti na dva načina: tako, da sestavljaš nov seznam, v katerega dodajaš elemente, in tako, da narediš kopijo seznama in iz nje brišeš prevečkratne ponovitve. Prvo je lažje, drugo poučnejše.

174. Brez n-tih

Napiši funkcijo `brez_ntih(s, n)`, ki iz podanega seznama s pobriše vsak n-ti element. Če pokličemo funkcijo z n=3, mora pobrisati elemente z indeksi 2, 5, 8, 11 in tako naprej.

```
>>> cc = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> brez_ntih(cc, 3)
>>> cc
[1, 2, 4, 5, 7, 8, 10]
```

Pazi: funkcija mora spreminjati podani seznam. Funkcija ne vrača nobenega rezultata!

175. Vse črke

Napiši funkcijo `vse_crke(beseda, crke)`, ki pove, ali množica `crke` vsebuje vse črke, ki se pojavijo v podani besedi. Pri tem sme množica vsebovati tudi črke, ki se v besedi ne pojavijo.

```
>>> vse_crke("AMFITEATER", set(["A", "M"]))
False
>>> vse_crke("AMFITEATER", set(["A", "M", "F", "I", "T", "E"]))
False
>>> vse_crke("AMFITEATER", set(["A", "M", "F", "I", "T", "E", "R"]))
True
>>> vse_crke("AMFITEATER", set(["A", "M", "F", "I", "T", "E", "R", "X", "O", "B",
"L"]))
True
```

176. Skritopis

Napiši funkcijo `skritopis(s)`, ki (kar predobro) skrije podano besedilo tako, da vsako besedo zamenja z njeno prvo črko. Vse ostale znake (presledke, ločila) pusti, kot so. Funkcija naj kot argument sprejme niz in kot rezultat vrne skriti niz.

```
>>> skritopis("Napiši funkcijo skritopis(s), ki (kar predobro) skrije besedilo
tako, da vsako besedo zamenja z njeno prvo črko.")
'N f s(s), k (k p) s b t, d v b z z n p č.'
```

Namig: četudi morda poznaš metodo `split`, je ne uporabi, saj ne naredi ničesar koristnega, razen če želiš najprej rešiti poenostavljeno nalogo, ki predpostavi, da v besedilu ni ločil, le besede in presledki med njimi.

177. Igra z besedami

Neki otroci se takole igrajo z besedami:

- Uredijo črke po abecedi. Tako iz HANA nastane AANH.
- Zamenjajo večkratne ponovitve črke s črko in številko, ki pove, kolikokrat se črka ponovi. Iz AANH tako dobijo A2N1H1.

Njihov postopek torej predela besedo HANA v A2N1H1.

Napišite funkcijo `skrij(beseda)`, ki takole predeluje besede. Kot argument dobi besedo (npr. `"HANA"`) in kot rezultat vrne predelano besedo (`"A2N1H1"`).

Pri tem ni potrebno, da funkcija izvaja natančno gornji postopek. Uberete lahko bližnjico, pomembno je le, da je rezultat enak. Prav lahko pride kaj (ne pa vse) od naslednjega: `defaultdict`, `Counter`, `sorted`.

178. Srečanje čebel

Ob neki potki so posejane rože. Količina nektarja v njih je opisana s seznamom. Obiranja se lotita dve čebeli: prva gre z leve proti desni, druga z desne proti levi. Čebela potrebuje eno sekundo, da se premakne na cvet in eno sekundo, da za obiranje vsake enote nektarja na cvetu. Napiši funkcijo `srecanje(vrt)`, ki pove, na katerem cvetu se bosta čebeli srečali. Za vrt `[1, 4, 1, 2, 8, 3]` mora vrniti 4.

Funkcija mora upoštevati, da se lahko čebeli srečata tudi med letom in ne samo na cvetu. Za seznam `[0, 0, 0, 0]`, naj vaš program vrne `1.5`.

Lahko se zgodi, da ob trenutku srečanja ostane kakšen cvet neobran. Čebeli se na vrtu `[6, 0]` srečata na prvem cvetu, na katerem je v tem trenutku še 5 enot nektarja.

Nasvet: najprej razmisli, koliko časa bosta potrebovali obe čebeli skupaj, nato za eno od čebel izračunajte, kje bo takrat.

179. Odvečni presledki

Med nami živijo tudi površneži, ki v besedilu puščajo odvečne   presledke med besedami. Napiši funkcijo `prepresledki(s)`, ki prejme niz in izračuna delež presledkov, ki so brez potrebe "podaljšani". "Podaljšan presledek" je vsako zaporedje presledkov, ki vsebuje dva ali več presledkov; pri tem se četverni presledek šteje kot en podaljšani presledek.

```
>>> prepresledki("Tule je      samo en preveč.")
0.25
>>> prepresledki("Tule ni nobenega preveč.")
0.0
>>> prepresledki("Ta   je    res   pretiran.")
1.0
>>> prepresledki("Problem!")
0.0
```

180. Kričiš!

Napiši funkcijo, ki kot argument prejme niz in vrne `True`, če ta vsebuje (vsaj) dve zaporedni veliki črki.

Namig: niz ima metodo `isalpha`, ki pove ali je niz sestavljen iz samih črk, in metodo `upper`, ki pove, ali so vse črke, ki se pojavijo v nizu, velike. Seveda ju ne uporabi na celotnem nizu, saj nas ne zanima, ali je celoten niz iz samih velikih črk, temveč ali vsebuje dve zaporedni veliki črki.

181. Napadalne kraljice

Šahovska polja označimo s črkami a-h, ki predstavljajo koordinate stolpcev, in številkami 1-8, ki predstavljajo vrstice. Tako je polje b4 drugo polje v četrti vrsti. Na šahovnico postavimo določeno število kraljic. Razpored predstavimo s seznamom koordinat, ki so opisane z nizi, recimo `["a4", "c7", "d2"]`.

Šahovska kraljica se lahko premika po stolpcih, vrsticah in diagonalah. Kraljica, ki bi jo postavili na b4, bi s svojo požrešnostjo ogrožala vsa polja stolpca b, vrste 4, poleg tega pa še diagonali, torej polja a5, c3, d2, e1 in a3, c5, d6, e7 in f8.

Napiši naslednje funkcije:

- `stolpec_prost(stolpec, razpored)` vrne `True`, če v danem stolpcu ni nobene kraljice;
- `prosti_stolpci(razpored)` vrne seznam stolpcev, v katerih ni nobene kraljice;
- `prost_stolpec(razpored)` vrne prvi stolpec, v katerem ni nobene kraljice; če takšnega stolpca ni, vrne `None`;
- `napada(polje1, polje2)` vrne `True`, če se polji napadata in `False`, če se ne; dogovorimo se, da polje napada tudi samo sebe;
- `napadajo(polje, razpored)` vrne seznam kraljic, ki napadajo podano polje;
- `napadeno(polje, razpored)` vrne `True`, če je podano polje napadeno;
- `prosto_v_stolpcu(stolpec, razpored)` vrne seznam nenapadenih polj v stolpcu;
- `napadajoce_se(razpored)` vrne vse pare kraljic, ki se napadajo med sabo;
- `legalen(razpored)` vrne `True`, če razpored vsebuje osem kraljic, ki se ne napadajo med sabo.

182. Obljudeni stolpci

Napiši funkcijo `po_stolpcih(s)`, ki prejme seznam šahovskih polj, na katerih stojijo figure in vrne seznam z osmimi števili, ki povedo, koliko figur stoji v posameznem stolpcu.

Poleg tega napiši funkcijo `naj_stolpec(s)`, ki pove, kateri je najbolj zasedeni stolpec. Če je več najbolj zasedenih stolpcev več, naj vrne prvega po abecedi.

Pomoč: funkcija `ord(c)` vrne kodo znaka (`ord("a")` vrne `97`), funkcija `chr(i)` pa vrne znak z določeno kodo (`chr(97)` vrne `"a"`).

183. Banka

Banka ima transakcije strank shranjene kot seznam parov parov (ime osebe, znesek). Tako seznam `[("Ana", 2), ("Berta", 8), ("Ana", 4), ("Berta", -3)]` pomeni, da je Ana vložila dva cekina, Berta 8, nato Ana 4, na koncu pa je Berta dvignila 3 cekine iz banke.

Napiši naslednje funkcije:

- `klienti(transakcije)` vrne seznam imen klientov; vsako ime naj se pojavi le enkrat. Vrstni red naj bo enak vrstnemu redu imen, ki se pojavljajo v transakcijah;

- `bilanca(transakcije, ime)` vrne, koliko denarja ima v banki oseba `ime` po vseh transakcijah v seznamu transakcije (v začetku ni imela ničesar). Če se ime v seznamu transakcij ne pojavi, oseba nima denarja.
- `najbogatejši(transakcije)` vrne ime najbogatejšega klienta; predpostaviti smeš, da ima banka vsaj enega klienta, ki ima več kot 0 cekinov;
- `racunovodja(transakcije)` vrne seznam z imeni klientov in njihovim premoženjem. Da bo lažje, naj vrnjeni seznam ne vsebuje terk. Tako, recimo, klic `racunovodja([("Ana", 2), ("Berta", 8), ("Ana", 4), ("Berta", -3)])` vrne `[["Ana", 6], ["Berta", 5]]`.

184. Srečni gostje

Za okroglo mizo sedijo (okrogli?) gosti. Nekateri so srečni, drugi ne. Konkretno: srečni so moški, ki sedijo med dvema ženskama in ženske, ki sedijo med dvema moškima.

Razpored gostov podamo s seznamom njihovih številk EMŠO. Napiši funkcijo `stevilo_srecnezev(razpored)`, ki prejme takšen seznam in vrne število srečnih gostov.

Pazi: ker je miza okrogla, sedi prvi gost poleg zadnjega.

Če sedita moški in ženska sama za okroglo mizo, smemo predpostaviti, da sta srečna; navsezadnje ima v tem primeru ženska na vsaki strani enega (in istega) moškega in obratno.

Primer (srečneži so podčrtani):

```
>>> stevilo_srecnezev(['0903912505707', '0110913506472', '2009956506012',
'1102946502619', '1902922506199', '2602930503913', '0204940508783',
'1602960505003', '0207959502025', '0207962509545'])
4
>>> stevilo_srecnezev(['1012947507186', '0506929507291', '3107910505475',
'1109984500497', '0510960506179', '0307978501042', '1607944505399',
'1706954501918', '1305934508423', '1406967504211'])
7
>>> stevilo_srecnezev(['0503973503512', '3004964501773', '1005933505567',
'2905936507573', '0712966507144'])
0
>>> stevilo_srecnezev(['0702948501362', '1505987508785'])
2
>>> stevilo_srecnezev(['2807955501835', '1604923501254', '0601925504908'])
0
>>> stevilo_srecnezev(['2807955501835'])
0
>>> stevilo_srecnezev([])
0
```

185. Gostoljubni gostitelji

Napiši funkcijo `razporedi(gosti)` za načrtovanje srečnih zabav. Funkcija dobi seznam gostov (spet kot seznam številk EMŠO) in mora vrniti seznam, ki je preurejen tako, da bodo gostje čim srečnejši. Pri tem se lahko zgodi, da je moških več kot žensk ali obratno, ali pa celo, da so na zabavi le moški ali le ženske.

Da ne bi predolgo ugibali očitnega: idealni razpored dobimo, če postavimo moške v eno kolono, ženske v drugo, potem izmenično jemljemo iz vsake po enega in na koncu posedemo nesrečne pripadnike tistega spola, ki ostane.

Primer (rešitve niso enolične – enako srečnost gostov je možno dobiti tudi z drugačnimi razporedi!)

```
>>> razporedi(['0505913509174', '2202973506004', '0304943506069', '2702943501809',
'2407980508463', '0209965503761', '2109913502875', '1802924506701',
'0207970500808', '1501917509568'])
['2702943501809', '0505913509174', '0209965503761', '2202973506004',
'2109913502875', '0304943506069', '0207970500808', '2407980508463',
'1802924506701', '1501917509568']
>>> razporedi(['2806984508656', '0602925509884', '1102979507594', '1104915509537',
'1502929506188', '1104923504226'])
['1104923504226', '2806984508656', '0602925509884', '1102979507594',
'1104915509537', '1502929506188']
>>> razporedi(['2806984508656', '0602925509884', '1102979507594',
'1104915509537'])
['2806984508656', '0602925509884', '1102979507594', '1104915509537']
>>> razporedi(['2806984508656'])
['2806984508656']
>>> razporedi([])
[]
```

186. Po starosti

Napiši funkcijo `po_starosti(s)`, ki dobi kot argument seznam `s`, ki vsebuje pare (ime osebe, EMŠO) in vrne nov seznam, ki vsebuje imena oseb, urejena po starosti, začenši z najstarejšimi.

EMŠO je sestavljena tako, da sta prvi števki rojstni dan v mesecu, drugi števki predstavljata mesec, naslednje tri pa leto brez tisočice. Prvih sedem števk EMŠO osebe, rojene 10. 5. 1983, bi bil 1005983. Prvih sedem števk EMŠO osebe, rojene 2. 3. 2001, pa bi bilo 0203001. Ali je oseba rojena leta 1xxx ali 2xxx, sklepamo po stotici.

```
>>> osebe = [("Ana", "2401983505012"), ("Berta", "1509980505132"),
             ("Cilka", "0203001505333"), ("Dani", "1005983505333")]
>>> po_starosti(osebe)
['Berta', 'Ana', 'Dani', 'Cilka']
```

187. Ujeme

Napiši funkcijo `ujeme(b1, b2)`, ki kot vhod prejme dve besedi in vrne novo besedo, ki vsebuje tiste črke, v katerih se besedi ujemata, črke na ostalih mestih pa so zamenjane s pikami. Če

besedi nista enako dolgi, naj bo nova beseda toliko dolga, kolikor je dolga krajša izmed podanih besed.

```
>>> ujeme("ROKA", "REKE")
'R.K.'
>>> ujeme("ROKA", "ROKAV")
'ROKA'
>>> ujeme("CELINKE", "POLOVINKE")
'..L....'
```

188. Najboljše prileganje podniza

Napiši funkcijo `naj_prileg(s, sub)`, ki v nizu `s` poišče mesto, ki se mu niz `sub` najbolj prilega. Funkcija naj vrne indeks v nizu `s`, število ujemajočih se črk in podniz znotraj `s`.

```
>>> naj_prileg("Poet tvoj nov Slovencem venec vije", "vine")
(24, 3, 'vene')
```

Besede "vine" v nizu ni, pač pa se z njim najbolj ujeme "vene", ki se nahaja na 24 znaku, saj se ujemata v treh črkah. Zato je rezultat funkcije `(24, 3, 'vene')`.

Kadar se enako dobro prilega več mest, naj funkcija vrne prvo izmed njih.

189. Deljenje nizov

Nize v Pythonu lahko pomnožimo s številom: `"bla"*3` vrne `"blablabla"`. Napiši funkcijo `deli_niz(s, k)`, ki niz `s` "deli" s številom `k`. Torej: `deli_niz(s, k)` mora vrniti tak niz, da bi, če ga pomnožimo s `k`, spet dobili `s`. Če niz ni sestavljen iz `k` ponovitev nekega niza, naj funkcija vrne `None`.

```
>>> deli_niz('toktoktoktok', 4)
tok
>>> deli_niz('tktktktk', 2)
tktk
>>> deli_niz('XXX', 3)
X
>>> print(deli_niz('toktoktoktok', 3))
None
>>> print(deli_niz('tiktoktak', 3))
None
```

190. Bralca bratca

Brata Peter in Pavel morata za domače branje prebrati vse knjige na knjižni polici. Nepridiprava sta se odločila, da jih bo vsak prebral približno pol (po številu strani), o drugi polovici pa se bo pustil poučiti od brata: Peter bo bral knjige z leve strani police, Pavel z desne. Sestavi funkcijo `razdeli_knjige(debeline)`, ki kot argument dobi seznam z debelinami knjig in vrne par (terko), ki pove, koliko knjig naj prebere eden in koliko drugi.

```
>>> razdeli_knjige([500, 100, 100, 100, 900])
(4, 1)
>>> razdeli_knjige([500, 100, 100, 100, 900, 100])
(4, 2)
>>> razdeli_knjige([50, 86, 250, 13, 205, 85])
(3, 3)
>>> razdeli_knjige([50, 86, 150, 13, 205, 85])
(4, 2)
>>> razdeli_knjige([5, 5])
(1, 1)
>>> razdeli_knjige([5])
(0, 1)
>>> razdeli_knjige([])
(0, 0)
```

Prvi primer razkriva namen vaje. Peter prebere štiri knjige, ki imajo skupaj 800 strani, Pavla pa doleti Dostojevski in ta odtehta vse štiri Petrove knjige. V drugem primeru dobi Pavel še eno knjigo; čeprav bi bilo pravičneje, če bi jo Peter (in bi imela vsak 900 strani), smo pač dogovorjeni, da eden bere z leve drugi z desne.

Namig: ne ukvarjaj se s Pavlom, temveč le s Petrom: koliko knjig mora prebrati, da jih bo približno pol? Pri "srednji knjigi" preveri, ali bomo bližje polovici, če jo prebere (in morda malo preseže polovico) ali jo pusti Pavlu (in prebere malo manj kot pol).

191. Turnir Evenzero

Turnirska pravila starodavne angleške igre Evenzero so naslednja:

- na turnirju se vedno pomeri 2^n (torej 1, 2, 4, 8, 16...) tekmovalcev,
- v vsakem krogu se pomeri prvi tekmovalec z drugim, tretji s četrtim, peti s šestim in tako naprej. Zmagovalci izmed teh parov se uvrstijo v naslednji krog.

Igra je preprosta: tekmovalca preštejeta, kolikšna je vsota dolžin njunih imen. Če je soda, zmaga prvi tekmovalec, če liha, drugi. Napišite funkcijo, ki kot argument prejme seznam tekmovalcev in kot rezultat vrne ime zmagovalca.

```
>>> turnir(['Alice', 'Bob', 'Tom', 'Judy'])
'Judy'
```

V prvem krogu tekmujeta Alice in Bob (zmaga prvi, torej Alice, ker je vsota dolžin njunih imen, osem, soda) in Tom in Judy (zmaga drugi, Judy, ker je dolžina imen liha). V drugem krogu tekmujeta Alice in Judy; zmaga drugi, Judy, saj je dolžina liha.

192. Najdaljše nepadajoče zaporedje

Napiši funkcijo, ki v seznamu poišče najdaljše nepadajoče zaporedje in vrne njegovo dolžino. Nekaj primerov:

```
[1, 2, 3, 2, 3, 4, 5]
```

V seznamu `l` sta skriti dve nepadajoči zaporedji, `[1, 2, 3]` in `[2, 3 , 4, 5]`, torej je odgovor 4.

```
[1, 2, 6, 8, 10, 1, 1, 1, 1, 1, 1, 0]
```

Seznam `l` tokrat razpade na nepadajoča zaporedja [1, 2, 6, 8, 10], [1, 1, 1, 1, 1, 1] in [0]. Odgovor je 6.

```
[1, 4, 6, 2, 3, 4, 7, 1, 4, 5, 8, 2, 4, 1, 4, 5, 5, 6, 8, 7, 3, 5, 5, 3, 5, 7, 1,
2, 2, 5, 7, 9, 9, 9, 1, 2]
```

Odgovor je 8.

193. Seznam vsot seznamov

Podan je seznam seznamov. Napiši naslednji funkciji.

Prva, `vsota_seznamov`, naj za vsak podseznam izračunaj njegovo vsoto in zloži vsote v nov seznam. Tako naj za, na primer, seznam `[[2, 4, 1], [3, 1], [], [8, 2], [1, 1, 1, 1]]` naračuna [`7, 4, 0, 10, 4]`, saj je `2 + 4 + 1 = 7`, 3 + 1 = 4 in tako naprej.

Druga, `najvecja_vsota`, naj poišče in vrne seznam z največjo vsoto elementov. Tako za gornji seznam `vrne [8, 2]`.

194. Veliko, a ne več kot

Napiši funkcijo `naj_pod(s, n)`, ki kot argument dobi seznam `s`, ki vsebuje pozitivna števila, in vrne podseznam z največjo vsoto, ki je še manjša ali enaka `n`. Če je seznamov z enako največjo vsoto več, naj vrne prvega med njimi.

```
>>> naj_pod([2, 1, 5, 6, 11, 2, 3, 6], 16)
[11, 2, 3]
>>> naj_pod([2, 1, 5, 6, 11, 2, 3, 6], 3)
[2, 1]
>>> naj_pod([2, 1, 5, 6, 11, 2, 3, 6], 28)
[1, 5, 6, 11, 2, 3]
```

Namig: z zanko pojdi prek vseh možnih začetkov podseznamov in znotraj tega prek vseh možnih koncev (med tem začetkom in koncem). Za vsak takšen podseznam preveri, ali ima vsoto, ki je večja od največje doslej, a manjša ali enaka `n`, in si, če je tako, zapomni vsoto, začetek in konec.

195. Nepadajoči podseznami

Napiši funkcijo, ki kot argument prejme seznam in kot rezultat vrne seznam sestavljen iz nepadajočih podseznamov. Tako mora za seznam `[2, 5, 7, 8, 4, 6, 9, 14, 7, 8, 3, 2, 5, 6]` vrniti seznam seznamov `[[2, 5, 7, 8], [4, 6, 9, 14], [7, 8], [3], [2, 5, 6]]`.

196. Sodi vs. lihi

Napiši funkcijo `sodi_vs_lihi(s)`, ki kot argument dobi seznam števil. Funkcija naj ugotovi, ali je v seznamu več sodih ali lihih števil in vrne seznam tistih, ki jih je več. Če je obojih enako, naj vrne vsa soda števila.

```
>>> sodi_vs_lihi([1, 2, 7, 14, 33, 140, 5])
[1, 7, 33, 5]
>>> sodi_vs_lihi([1, 2, 7, 4])
[2, 4]
>>> sodi_vs_lihi([])
[]
```

197. Gnezdeni oklepaji

Imamo izraz, ki je zapisan kot niz, sestavljen samo iz oklepajev in zaklepajev. Napišite funkcijo, ki ugotovi regularnost takega izraza, torej, če se vsi oklepaji ustrezno zaprejo z zaklepaji.

Tile izrazi so pravilni "`()`", "`(())`", "`(()()())()`", tile pa ne: "`((`", "`)(`", "`(()()))`", "`(()))(`".

Zdaj pa napiši "zaresno" funkcijo, ki preveri oklepaje v nizih, ki vsebujejo cele račune. Pri tem naj opazuje samo oklepaje; kar je med njimi, sme biti popolnoma napačno. Primeri pravilnih izrazov so torej "1+1", "2*(3+(-6))" in tudi "((2+3)*(4+)-abc/4&&&!'#$!3)", saj opazujemo le oklepaje.

198. Črkovni oklepaji

Rešimo podobno nalogo kot so bili Gnezdeni oklepaji, a vzemimo namesto oklepajev črke: velike črke naj predstavljajo "oklepaje", male črke pripadajoče "zaklepaje". Tokrat moramo poleg tega, koliko oklepajev in zaklepajev imamo, paziti tudi na to, za katero črko gre: veliki A "zapremo" le z malim a, ne pa tudi z malim b ali velikim A. Na primer, niz "ABba" je pravilen, a "ABab" pa ne, ker bi se moral B zapreti pred A. Pravilni oklepaji so tudi, "Aa", "AaBb", "AaBbACBbDdcDda", primeri nepravilnih pa "ABC", "aB", "AA", "aA", "ABCabc". (Kdor pozna zapis HTML, se je morda spomnil na oznake v njem. Oznake v pravilnem HTMLju se tudi pravilno zaključujejo.)

Napiši funkcijo, ki prejme takle niz in pove, ali je pravilen ali ne.

Rešitev je kratka, vendar lahka ali težka, odvisno od tega, ali vemo, kaj je sklad. Kdor ne ve, naj jo raje pusti. Kdor ve, pa bo za sklad uporabil kar seznam. Pythonovi seznami imajo metodo `pop`, za `push` pa nam služi `append`.

199. Brez oklepajev

Napiši funkcijo, ki prejme niz in vrne nov niz, v katerem je pobrisano vse besedilo, ki se nahaja v oklepajih (skupaj z oklepaji). Če dobi kot argument niz "`Tole ima (nekaj) besedila (v oklepaju).`" naj vrne "`Tole ima  besedila .`" Predpostaviti smeš, da vsakemu oklepaju sledi zaklepaj in da ni zaklepajev brez oklepajev.

Znaš napisati tudi funkcijo, ki dovoljuje tudi gnezdene oklepaje (kot jih imamo (recimo tule) v tem stavku)? Za ta stavek mora funkcija vrniti `"Znaš napisati tudi funkcijo, ki dovoljuje tudi gnezdene oklepaje ?"`

200. Vsota kvadratov palindromnih števil

Število je palindrom, če se naprej bere enako kot nazaj; primer takšnega števila je 737. Napišite program, ki izpiše vsoto kvadratov vseh palindromnih števil, ki so manjša od 1000.

201. Skupinjenje

Napiši funkcijo `group(s)`, ki v podanem seznamu `s` združi zaporedne enake elemente v podseznаme.

```
>>> group([1, 1, 3, 3, 2, 2, 2, 2, 1, 1, 2, 2, 2])
[[1, 1], [3, 3], [2, 2, 2, 2], [1, 1], [2, 2, 2]]
>>> group([1, 2, 3, 4, 5])
[[1], [2], [3], [4], [5]]
```

202. Trdnjava

Figuro prestavljamo po neskončni šahovnici. Premik opišemo z nizom, dolgim dva znaka; prvi je črka U, D, L ali R, ki predstavlja smer (up, down, left in right) in drugi je številka med 0 in 9, ki pove, za koliko polj prestavimo figuro v podani smeri. Začetna pozicija je (0, 0) - uporabljali bomo kar takšno notacijo, ne "šahovske".

Napiši funkcijo `trdnjava(s)`, ki kot argument dobi seznam potez in vrne `True`, če se trdnjava kadarkoli med premikanjem (ne nujno na koncu!) ponovno znajde na izhodiščnem polju (0, 0) in `False`, če se ne.

```
>>> trdnjava(['U5', 'D5'])
True
>>> trdnjava(['U5', 'D6']) # trdnjava gre PREK polja (0, 0), a ne nanj!
False
>>> trdnjava(['R5', 'L5', 'R3'])
True
>>> trdnjava(['R5', 'U4', 'L3'])
False
>>> trdnjava(['L3', 'U2', 'D2', 'R1', 'R1'])
False
>>> trdnjava(['U0'])
True
>>> trdnjava(['U0', 'U1'])
True
```

203. Vsote peterk

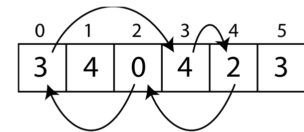
Podan je seznam nenegativnih števil in naloga je poiskati največjo vsoto petih zaporednih števil v tem seznamu. V seznamu [9, 1, 2, 3, 8, 9, 0, 4, 3, 7] se nam najbolj splača sešteti števila 3, 8, 9, 0 in 4 ali pa 8, 9, 0, 4 in 3. Odgovor je torej 24.

Tisti, ki se vam to zdi prelahko, lahko program preskusite na seznamu s sto tisoč elementi in z njim iščite največjo vsoto deset tisoč zaporednih števil.

204. Legalni konj

Napiši funkcijo, ki kot argument prejme koordinate konja na šahovnici (npr. B2) in vrne seznam vseh polj, na katera lahko konj skoči. Konj skače v obliki črke L: za dve polji navzgor ali navzdol in eno levo ali desno, ali pa za eno polje navzgor ali navzdol in za dve levo ali desno. Seveda ne sme skočiti ven iz šahovnice.

205. Skoki



Napiši funkcijo `skoki(s)`, ki prepotuje podani seznam števil `s`. Recimo, da dobi kot argument seznam `[3, 4, 0, 4, 2, 3]`. Funkcija vedno začne na ničtem polju. Ta v našem primeru vsebuje `3`, zato skoči na tretje polje. Na tretjem polju je `4`, torej skoči na četrto polje. Vsebina četrtega polja je `2`, zato skoči na drugo polje. To vsebuje `0`: spet smo na začetku, zato končamo.

Funkcija naj kot rezultat vrne število skokov do vrnitve na polje 0; v gornjem primeru je to `4`.

Poleg tega naj funkcija pazi še na dve možnosti:

- Če se slučajno zgodi, da kako polje, *na katerega pride*, vsebuje preveliko število, naj se funkcija ustavi in vrne `-1`. Če so v seznamu prevelike številke, vendar je pot ne pripelje nanje, ni nič narobe.
- Če se funkcija zacikla – najpreprostejši primer je seznam `[1, 1]`, kjer s prvega polja stalno skače na prvo polje – naj vrne `-2`.

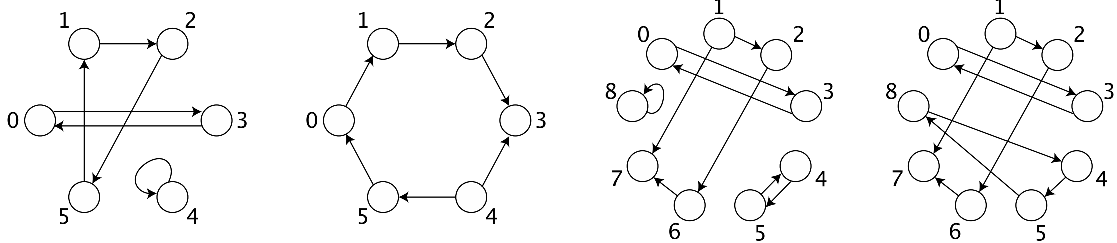
Namig: če se vrnemo na začetno polje, za to potrebujemo največ toliko skokov, kolikor je dolg seznam.

Predpostaviti smeš, da seznam ni prazen in vsebuje samo cela pozitivna števila.

```
>>> skoki([3, 4, 0, 4, 2, 3])
4
>>> skoki([1, 1]) # stalno ponavlja polje 1
-2
>>> skoki([1, 2, 3, 4, 5, 3]) # ponavlja 4, 5, 3, 4, 5, 3, 4, 5, 3 ...
-2
>>> skoki([1, 2, 3, 8]) # pride do 8 in "skoci ven"
-1
>>> skoki([0])
1
```

206. Bobri plešejo

Bobri plešejo nekakšno kolo. Postavijo se na polja v krogu. Iz vsakega polja je narejena puščica v neko drugo polje, ki pove, kam mora v naslednjem koraku bober, ki stoji na tam polju. V vsako polje pelje le ena puščica.



Posamezen ples bomo opisali s seznamom. Gornjim štirim ustrezajo

```
ples1 = [3, 2, 5, 0, 4, 1]
ples2 = [1, 2, 3, 4, 5, 0]
ples3 = [3, 2, 6, 0, 5, 4, 7, 1, 8]
ples4 = [3, 2, 6, 0, 5, 8, 7, 1, 4]
```

`ples1`, recimo, pravi, da gre bober iz ničtega polja na tretje, iz prvega na drugo, iz drugega na peto, iz tretjega na ničto in tako naprej.

Napišite `do_doma(ples, polje)`, ki kot argument prejme `ples` in številko polja. Vrniti mora število korakov, po katerem je plesalec, ki začne na podanem polju, spet na svojem polju. Tako mora, recimo `do_doma(ples1, 0)` vrniti 2 (plesalec gre najprej na polje 3 in potem nazaj na 0), `do_doma(ples1, 2)` pa vrne 3.

207. Bobri vsi domov

Napiši funkcijo `vsi_doma(ples)`, ki kot argument prejme ples v enakem zapisu kot v prejšnji nalogi, vrne pa število korakov, po katerem so *vsi* plesalci spet na svojih mestih.

208. Najbogatejši cvet

Rože v neki deželi, s katero se bomo ukvarjali v naslednjih nekaj nalogah, so posajene v pravokotno mrežo. Čebelam je pomembno, koliko nektarja je v posameznem cvetu. Ker so vešče programiranja, shranijo količino nektarja v seznamu seznamov, kot je tale.

```
[[1,3,3,8,5,4,2,1,5,6],
 [2,4,3,3,6,8,1,3,5,6],
 [4,5,6,4,7,4,3,6,4,7],
 [2,8,7,0,0,7,4,7,8,0],
 [2,3,4,7,0,8,7,6,3,8],
 [3,7,9,0,8,5,3,2,3,4],
 [1,5,7,7,6,4,2,3,5,6],
 [0,6,3,3,6,8,0,6,7,7],
 [0,1,3,2,8,0,0,0,0,0],
 [3,1,0,3,6,7,0,5,3,1],
 [1,3,5,7,0,8,6,5,3,1],
 [3,6,3,1,3,5,8,7,5,1],
 [4,3,6,0,0,8,4,7,5,3],
 [3,5,6,8,6,3,1,3,5,2]]
```

Vrtovi so lahko različnih velikosti - nekateri imajo tudi le po eno vrsto ali en stolpec rož, pa tudi količina nektarja v rožah je lahko poljubno pozitivno število. Lahko se zgodi celo, da ima vrt en sam cvet.

Koordinatni sistem, ki ga uporabljajo čebele, je takšen, da je točka (0, 0) v zgornjem levem (ne spodnjem levem!) vogalu. Ko gre čebela navzdol, koordinata y narašča: če čebela odleti, recimo, za eno vrstico nižje, se koordinata y poveča za 1.

Napiši funkcijo `naj_koordinate(vrt)`, ki vrne koordinato "najbogatejšega" cveta. V gornjem primeru mora funkcija vrniti `(2, 5)` (kot terko dveh int-ov), saj je tam cvet z 9 mg nektarja. Če sta na vrtu dva cveta z največjo količino nektarja, naj vrne tistega, ki je višje. Če sta dva na isti višini, naj vrne tistega, ki je bolj levo.

209. Pravilna pot

Čebele opisujejo pot z zaporedjem znakov "`L`", "`R`", "`U`" in "`D`" (kot *left*, *right*, *up* in *down*; angleške črke uporabljamo, ker se "desno" in "dol" v slovenščini začneta z isto črko). Pri tem D pomeni, da se koordinata y poveča za 1 (čebela gre eno vrstico nižje).

Obiranje cvetov se vedno začne pri cvetu (0, 0).

Napiši funkcijo `pravilna_pot(pot, vrt)`, ki prejme niz, ki opisuje `pot` (npr. "`RRDDRLL`") in `vrt`. Kot rezultat naj vrne `True`, če je pot pravilna in `False`, če ni. Pot ni pravilna, če čebela na njej kdaj zapusti vrt. Primeri nepravilnih poti za gornji vrt so, recimo "`DDRRRRRRRRRRRRRRRRRRRRRRR`" (čebelo odnese nekam na levo) in "`U`" (čebelo takoj odpihne navzgor).

210. Dobiček na poti

Napiši funkcijo `dobicek_na_poti(pot, vrt)`, ki pove, koliko nektarja bo čebela nabrala, če leta od cveta do cveta, kot ji veleva pot. Predpostaviti smeš, da je pot nikoli ne bo vodila na cvet, ki ga je že obrala.

Pot je tudi gotovo pravilna, tako da vsebuje le znake L, R, D in U ter nikoli ne vodi izven vrta.

Če na gornjem travniku preleti pot "`RDDL`" bo nabrala 17 mg nektarja (pri izračunu je potrebno upoštevati tako prvi kot zadnji cvet).

211. Enkratna pot

Napiši funkcijo `brez_ponovitve(pot)`, ki vrne `True`, če čebela, ki leti po podani poti, nikoli ne obišče dvakrat (ali večkrat) istega cveta.

212. Neobrani cvetovi

Količina nektarja v cvetovih enodimenzionalnega vrta (recimo potke) je opisana s seznamom števil, na primer `[5, 1, 3, 2, 5, 7, 3, 12, 2]`. Po vrtu (proti vsem pričakovanjem) leta čebela; vsakič, ko pride na cvet, pobere ves nektar v njem. Pot čebele je opisana s seznamom

pozitivnih in negativnih števil: `4` pomeni pomik za štiri polja desno, `-7` pomeni pomik za sedem polj levo. Čebela začne na polju `0` in se pomika, kot zahteva seznam. Pomikanje se ustavi ko

- je konec poti (pridemo do konca seznama korakov) ali pa
- čebela pride na cvet brez medu (ker ga je že obrala ali pa je bil prazen že na začetku).

Napiši funkcijo `ostanki(vrt, pot)`, ki simulira pot čebele in vrne količino nektarja, ki **ostane v vrtu**, ko se čebela ustavi. Predpostavi smeš, da pot ne vsebuje skokov prek meja vrta.

Funkcija ne sme spremeniti nobenega od podanih seznamov. (Namig: skopiraj seznam.)

213. Žabji skoki

Napiši funkcijo `zaba(skoki)`, ki kot argument prejme seznam parov smeri skokov in njihovih dolžin, kot rezultat pa vrne množico točk, ki jih je obiskala žaba, če je v začetku stala na koordinatah (0, 0). Tako mora, na primer, klic `zaba([("S", 5), ("Z", 2), ("V", 6), ("S", 3)])` vrniti množico `{(0, 0), (0, 5), (2, 5), (-4, 5), (-4, 8)}`.

Nato napiši funkcijo `zaba_znotraj(skoki, max_x, max_y)`, ki dobi množico skokov in vrne `True`, če je bila žaba ves čas znotraj pravokotnika med izhodiščem `(0, 0)` in `(max_x, max_y)`. Sicer naj vrne `False`.

Napiši tudi funkcijo `naj_razdalja(skoki)`, ki vrne razdaljo do najbolj oddaljene točke (torej, najbolj oddaljene od izhodišča).

214. Lov na muhe

Za žabo iz prejšnje naloge napiši funkcijo `muhe(koordinate, skoki)`, ki dobi množico s koordinatami muh (npr. `{(1, 0), (0, 0), (5, 0), (10, 10)}`) in seznam skokov. Na vsaki koordinati je največ ena muha. Žaba ima zelo kratek jezik in poje muho takrat (in natanko takrat), ko skoči nanjo. Funkcija naj vrne število pojedenih muh.

Napiši tudi funkcijo `pojedene(koordinate, skoki)`, ki vrne koordinate pojedenih muh in `prezivele(koordinate, skoki)`, ki vrne koordinate preživelih.

215. Žabja restavracija

Napiši funkcijo `zberi(narocila)`, ki dobi slovar naročil v restavraciji: ključi so imena strank, vrednosti so jedi, ki so jih stranke naročile. Funkcija naj prešteje, koliko katere jedi potrebujemo in to vrne v slovarju, katerega ključi so jedi in vrednosti količina teh jedi.

Tako mora klic `zberi({"Rega": {"mušji kraki", "pajek"}, "Kvakica": {"mušji kraki"}, "Dolgokrak": {"komarjev biftek"}})` vrniti `{"mušji kraki": 2, "pajek": 1, "komarjev biftek": 1}`.

216. Poštar iz Hamiltona

Nek duhovit poštar raznaša pošto izključno v neki zelo zelo dolgi ulici. Pri tem ne hodi od hiše do hiše po vrsti: odločil se je, da ne bo nikoli naredil poti med dvema hišama, ki si ne delita

vsaj ene skupne števke. Od hiše številka 75 lahko gre k hiši 78 (zaradi sedmice), 52 (zaradi petice), 107 (zaradi sedmice), ne pa k 43 ali 88, ki s 75 nimata nobene skupne števke. Seveda gre vsak dan le k hišam, za katere ima kako pošto.

Zaradi te čudne navade mora vsak dan dobro splanirati svojo pot, da bo raznesel vso pošto – pa še tako se mu kak dan zgodi, da ne more raznositi vse pošte, saj ni poti, ki bi ustrezala pravilom.

Napiši funkcijo `postar(naslovi)`, ki kot argument dobi seznam števil, h katerim mora prinesti pošto, kot rezultat pa vrne `True`, če bo lahko raznosil vso pošto in `False`, če ne.

Nasvet: mogoče ti bo lažje, če najprej napišeš funkcijo `preveri_zaporedje(zaporedje)`, ki dobi neko zaporedje števil in pove, ali je to pravilna pot v tem smislu, da si vsak zaporeden par števil deli vsaj eno števko. Potem napišete zahtevano funkcijo tako, da le-ta kliče `preveri_zaporedje(zaporedje)`.

Še en nasvet: toplo priporočam, da pred reševanjem preveriš, kaj izpiše program

```
from itertools import permutations
for x in permutations(["Ana", "Berta", "Cilka", "Dani"]):
    print(x)
```

217. Graf hamiltonskega poštarja

Napiši funkcijo `pari_his(stevilke)`, ki dobi seznam hišnih števil in vrne množico vse parov hiš, ki imajo vsaj eno skupno števko. Tako mora, recimo, `pari_his([12, 23, 26, 56])` vrniti `{(12, 23), (12, 26), (23, 12), (23, 26), (26, 12), (26, 23), (26, 56), (56, 26)}`. (Če v resnici vrne `{(26, 23), (12, 26), (23, 26), (23, 12), (12, 23), (26, 12), (26, 56), (56, 26)}`, se ne vznemirjaj, saj je to v resnici eno in isto. Vrstni red elementov množice ni pomemben.)

Nato napiši funkcijo `graf_his(pari)`, ki prejme seznam parov in vrne graf, zapisan kot slovar, katerega ključi so točke (številke), vrednosti pa množice točk, v katere je mogoče priti iz njih. Če jo pokličemo s pari `graf_stevilk({(12, 23), (12, 26), (23, 12), (23, 26), (26, 12), (26, 23), (26, 56), (56, 26)})`, mora vrniti `{26 : {23, 12, 56}, 23: {26, 12}, 12: {23, 26}, 56: {26}}`.

218. Pretakanje

Imejmo posodi A in B, ki držita 5 in 8 litrov. Poleg tega imamo ogromno posodo P, ki drži 1000 litrov. Posodi A in B sta v začetku prazni, v P pa je napol polna mleka. Med posodami bomo pretakali mleko tako, da bomo vedno napolnili posodo, v katero točimo, ali pa izpraznili posodo, iz katere točimo. Pretakanje A->B torej pomeni, da točimo mleko iz A v B toliko časa, da je B polna do roba ali pa da je A prazna.

Pretakanje mleka med posodami opišemo s seznamom nizov. Tako `["P->A", "A->B", "P->A", "A->B"]` pomeni, da najprej napolnimo posodo A iz posode P, nato pretočimo A v B (ker je B večja, bo šla vanjo vsa vsebina posode A), nato spet napolnimo A in spet točimo A v B. Ker je v

B že pet litrov, ki smo jih natočili prej, bodo šli vanjo le še trije litri mleka iz posode A in v A bosta na koncu ostala dva litra.

Napiši funkcijo `pretakanje(s)`, ki prejme seznam, podoben gornjemu in kot rezultat vrne količino mleka v prvi in v drugi posodi.

Predpostaviš smeš, da nikoli ne točimo iz iste posode v isto (na primer A -> A).

219. Slovar anagramov

Napiši funkcijo `slovar_anagramov(besede)`, ki dobi seznam besed in vrne slovar. Njegove *vrednosti* naj bodo množice besed, ki so sestavljene iz istih črk (tako kot besede `{"ravan", "ravna", "vrana"}`). Pripadajoči *ključi* naj bodo besede, sestavljene iz teh črk, a urejene po abecedi. V gornjem primeru bi bil ključ enak "aanrv".

```
>>> slovar_anagramov(["torba", "sloneti", "tesnilo", "botra", "obrat"])
{"abort": {"torba", "obrat", "botra"},
 "eilnost": {"sloneti", "tesnilo"}}
```

Poleg tega napišite funkcijo `anagrami(beseda, s)`, ki kot argument prejme besedo in slovar, kakršnega vrača zgornja funkcija. Vrniti mora množico anagramov te besede.

```
>>> poisci_anagrame("obrat", {"abort": {"torba", "obrat", "botra"},
                              "eilnost": {"sloneti", "tesnilo"}})
{"torba", "obrat", "botra"}
```

Namig: `"".join(sorted(x))` dela čudeže.

220. Človek ne jezi se

V poenostavljeni igri Človek ne jezi se ima vsak igralec eno figuro. Vsi začnejo na polju 0. Ko je igralec na vrsti, vrže kocko in premakne figuro za toliko polj, kolikor pokaže kocka. *Če pri tem pride na polje, na katerem že stoji kateri drugi igralec, gre oni, drugi igralec na polje 0.*

Napiši funkcijo `clovek_ne_jezi_se(igralcev, meti)`, ki kot argument dobi število igralcev in zaporedje metov, kot rezultat pa vrne številke polj, na katerih se po teh metih nahajajo igralci.

221. Največ dvakrat

Napiši funkcijo `najvec_dve(s)`, ki v podanem seznamu pusti do dve (ne nujno zaporedni) pojavitvi vsakega elementa in pobriše vse nadaljnje. Funkcija mora vrniti `None`; spreminja naj podani seznam.

Seznam ne vsebuje nujno števil, predpostaviti pa smete, da so njegovi elementi nespremenljivi (*immutable*).

Če imamo `s = [4, 1, 2, 4, 1, 3, 3, 1, 2, 5, 4, 3, 7, 4]`, mora biti po klicu `najvec_dve(s)` seznam `s` enak `[4, 1, 2, 4, 1, 3, 3, 5, 7]`. Briše namreč takole: [4, 1, 2, 4, 1, 3, 3, ~~1,~~ 2, 5, ~~4, 3,~~ 7, 4].

222. Seštej zaporedne

Napiši funkcijo `sestej_zaporedne(s)`, ki prejme seznam števil in vrne nov seznam, v katerem so vse zaporedne pojavitve istega števila zamenjane z vsoto. Tako mora, na primer, klic `sestej_zaporedne([1, 4, 4, 4, 0, 4, 3, 3, 1])` vrniti `[1, 12, 0, 4, 6, 1]` – tri štirice se seštejejo v 12 in dve trojki se seštejeta v 6. Ostalo ostane, kot je.

223. Intervali

Napiši funkcijo `intervali(s)`, ki prejme seznam števil in vrne seznam parov, ki predstavljajo začetke in konce intervalov naraščajočih zaporednih števil v tem seznamu. Tako mora klic `intervali([4, 5, 6, 7, 15, 21, 22, 23])` vrniti seznam `[(4, 7), (15, 15), (21, 23)]`.

Napiši še funkcijo `razpisi(ints)`, ki dela ravno obratno – prejme, recimo `[(4, 7), (15, 15), (21, 23)]` in vrne `[4, 5, 6, 7, 15, 21, 22, 23]`.

224. Dolžine ladij

Recimo, da položaj pri igri Potapljanje ladjic predstavimo s seznamom nizov, kot, recimo

```
[".......XXX....X",
 "X..XX.......X..",
 "X......XXXXX...",
 "X..XX........."]
```

Napiši funkcijo `dolzina(plosca, x, y)`, ki prejme takšno ploščo in pove, kako dolga je ladja, ki se začne na koordinatah `x`, `y` in gre potem na desno ali navzdol.

Poleg tega napiši funkcijo `najdaljsa_ladja(plosca)`, ki vrne dolžino najdaljše ladje na plošči; v gornjem primeru je to 5.

Premetavanje nizov in besedil

225. Sekunde

Ljudje čas radi zapisujemo v obliki, kot je, na primer, 8:12:34, kar bi, recimo, pomenilo 12 minut in 34 sekund čez osem ali pa 17:4:45, kar je štiri minute in 45 sekund čez peto popoldan. Računalniki čas raje merijo v sekundah – za potrebe te naloge, ga bomo merili v sekundah od polnoči. Napiši funkcije:

1. `cas_v_sekunde(s)`, ki prejme čas v »človeški« obliki in kot rezultat vrne čas od polnoči. Upoštevajte, da sme človek napisati bodisi `10:05:20` bodisi `10:5:20` – funkcija naj zna brati oboje.
2. `sekunde_v_cas(s)`, ki prejme čas v sekundah od polnoči in vrne niz v človeški obliki. Funkcija naj rezultat vrne v obliki `10:05:20`, ne `10:5:20`.
3. `razlika_casov(s1, s2)`, ki prejme dva časa v človeški obliki in vrne razliko med njima, spet zapisano v človeški obliki. Predpostaviti smeš, da je `s2` večji od `s1`.

Pri reševanju naloge je prepovedano uporabljati Pythonove module.

```
>>> cas_v_sekunde("10:00:00")
36000
>>> cas_v_sekunde("0:0:0")
0
>>> cas_v_sekunde("10:05:12")
36312
>>> sekunde_v_cas(36312)
'10:05:12'
>>> sekunde_v_cas(0)
'00:00:00'
>>> razlika_casov("10:00:00", "10:11:5")
'00:11:05'
>>> razlika_casov("10:00:00", "12:11:5")
'02:11:05'
>>> razlika_casov("10:11:30", "12:05:35")
'01:54:05'
```

226. Naslednji avtobus

Napiši funkcijo `naslednji_avtobus(prihodi)`, ki dobi seznam prihodov avtobusov in vrne "številko" (npr. `"1"` ali `"6b"`) avtobusa, ki bo prišel prvi. Če je za več avtobusov napovedan isti čas do prihoda, naj vrne tistega z nižjo številko (a pazite: avtobus 2 ima nižjo številko od avtobusa 11, prav tako ima avtobus 6b nižjo številko od 11). Prihodi avtobusov so podani kot slovar, katerega ključi so "številke" avtobusov (`"1"`, `"6b"`...), vrednosti pa seznami napovedanih časov do prihoda. Časi niso nujno urejeni po velikosti.

```
>>> naslednji_avtobus({"1": [5, 7], "3": [3, 11], "6b": [7]})
"3"
>>> naslednji_avtobus({"1": [5, 7], "2": [11, 3, 18], "11": [3, 7]})
"2"
>>> naslednji_avtobus({"1": [5, 7], "2": [11, 3, 18], "6b": [3, 7]})
"2"
```

227. Eboran

Napiši funkcijo `eboran(stavek)`, ki prejme stavek (zapisan kot besede, med katerimi so presledki, brez ločil) in vrne nov stavek, v katerem so vse besede obrnjene.

```
>>> eboran("vse je narobe tudi tale stavek")
esv ej eboran idut elat kevats
```

Pazi, `"kevats elat idut eboran ej esv"` ni pravilna rešitev naloge, vrstni red obrnjenih besed mora ostati enak.

Kaj pa ločila? Znate napisati funkcijo, ki bi obrnila le besede, ločila in presledka pa pustila pri miru?

```
>>> eboran2('zapel je: "vse je narobe, tudi     tale stavek."')
lepaz ej: "esv ej eboran, idut     elat kevats."
```

228. Cenzura

Napiši funkcijo `cenzura(besedilo, prepovedane)`, ki prejme besedilo in seznam prepovedanih besed. Vrniti mora nov niz, v katerem so vse prepovedane besede zamenjane s toliko X-i, koliko črk imajo. V množici prepovedanih besed so vse besede napisane z malimi črkami, vendar so prepovedane tudi, če so napisane z velikimi.

Predpostaviti smeš, da v besedilu ni nobenih ločil, le besede in presledki. Če znaš, pa napiši funkcijo, ki dela tudi z ločili in pusti stavek takšen, kot je, le prepovedane besede zamenja.

```
>>> prepovedane = {"zadnjica", "tepec", "pujs", "kreten"}
>>> cenzura("Pepe je ena navadna zadnjica in pujs in še kaj hujšega",
            prepovedane)
'Pepe je ena navadna XXXXXXXX in XXXX in še kaj hujšega'
>>> cenzura("Pepe je ena velika Zadnjica in PUJS in še kaj hujšega",
            prepovedane)
'Pepe je ena velika XXXXXXXX in XXXX in še kaj hujšega'
>>> cenzura("Pepe je okreten", prepovedane)
'Pepe je okreten'
```

229. Črkovalnik

Napiši funkcijo `crka(c, a)`, ki kot argument prejme dva niza dolžine 1. Prvi niz predstavlja črko in drugi "akcijo". Funkcija naj vrne naslednje:

- če je akcija enaka `"U"`, vrne veliko črko `c`: `crka("t", "U")` vrne `"T"`;
- če je akcija števka med `"0"` in `"9"`, vrne toliko ponovitev črke `c`: `crka("t", "3")` vrne `"ttt"`;
- če je akcija enaka `"x"`, vrne prazen niz: `crka("t", "x")` vrne prazen niz;

- če je akcija enaka ".", vrne nespremenjeno črko c: crka("t", ".") vrne "t".

Poleg tega napiši funkcijo kodiraj(beseda, koda), ki prejme besedo in akcije za posamezne črke, ter vrne novo besedo, v kateri je vsaka črka predelana tako, kot zahteva ustrezna črka iz akcije. Predpostaviti smeš, da sta niza beseda in koda enako dolga.

```
>>> kodiraj("beseda", ".U5.x.")
bEsssssea
```

230. Hosts

V Unixu (vključno z Linuxom in Mac OS X) se v imeniku /etc nahaja datoteka hosts. Na MS Windows jo najdemo v c:/Windows/System32/drivers/etc. Videti je lahko, recimo, takole

```
# localhost name resolution is handled within DNS itself.
# 127.0.0.1 localhost
# ::1 localhost
193.2.72.47 google.com # bolj zadane!
193.2.72.47 www.google.com # bolj zadane!
193.2.72.35 bing.com
193.2.72.35 www.bing.com
```

Pravila za branje so takšna:

1. če je v vrstici kakšen #, odbijemo vse od njega do konca;
2. pobrišemo ves beli prostor (presledki, tabulatorji...) na začetku in koncu;
3. če ne ostane nič več, vrstico ignoriramo;
4. sicer pa ostaneta v vrstici dve stvari: IP in ime nekega strežnika.

Sestavite funkcijo beri_hosts(), ki prebere tako oblikovano datoteko hosts (nahaja naj se kar v trenutnem direktoriju) in vrne slovar, katerega ključi so imena, vrednosti pa IPji.

231. Uporabniške skupine

Unix zapisuje skupine uporabnikov v datoteko /etc/group. Vsaka vrstica opisuje eno skupino uporabnikov. V njej so štirje podatki, ločeni z dvopičji: ime skupine, geslo, številski id in seznam uporabnikov v skupini. Uporabniška imena so ločena z vejicami. Del datoteke je lahko, recimo, takšen.

```
ana:x:500:ana
berta:x:501:berta
cilka:x:502:cilka
dani:x:503:dani
ordinary:x:504:ana,berta,cilka,dani
taccess:x:505:berta,cilka
wusers:x:506:ana,cilka
```

V njej vrstica

```
taccess:x:505:berta,cilka
```

pravi, da sta v skupini `taccess` uporabnici `berta` in `cilka`. Geslo in številka sta za to nalogo nepomembna.

Napiši funkcijo `beri_skupine(ime_dat)`, ki kot argument prejme ime datoteke `groups` in kot rezultat vrne slovar, katerega ključi so uporabniki, vrednosti pa skupine, v katerih je ta uporabnik (torej obratno od tega, kar je v `groups`, kjer za skupino izvemo, katere uporabnike vsebuje).

```
>>> users = beri_skupine("group")
>>> users["ana"]
['ana', 'ordinary', 'wusers']
>>> users["cilka"]
['cilka', 'ordinary', 'taccess', 'wusers']
>>> users["dani"]
['dani', 'ordinary']
```

232. Skrito sporočilo

Napiši program, ki bo dešifriral sovražnikova sporočila. Iz zanesljivih virov si izvedel, da sovražnik uporablja nadvse preprosto šifro. Vse, kar moraš storiti, je izpisati vse znake, ki se pojavijo na začetku kakšne povedi. Drugače rečeno, izpisati moraš prvi znak niza in vse znake, ki sledijo nizu `'. '` (in, pazi, nekatere povedi se začnejo s presledkom).

V kaj se dešifrira naslednje sporočilo?

```
Nic ne bo. Ana bo ostala doma. Prisla je njena mama. Aleksa tudi
ne bo. Dejan je zbolel.  Groza. Ostali smo samo se jaz, ti in
Miha. Bolje, da izlet prestavimo.  Pa tako sem se ga veselil. 6 dni
sem cakal na to, da se odpravimo v hribe. Hja, pa drugic...
```

233. Iskanje URLjev

URL je niz, ki se (za potrebe te naloge) začne s `http://` ali `https://` in nadaljuje vse do prvega presledka, tabulatorja ali znaka za novo vrsto. Vsebuje lahko torej tudi poljubne znake, kot so oklepaji, pike, plusi in minusi ... karkoli razen belega prostora. Napiši funkcijo `najdi_URLje(s)`, ki kot argument prejme niz in kot rezultat vrne seznam URLjev v njem.

```
>>> s = """O regularnih izrazih lahko preberes v Pythonovi -
http://www.python.org - dokumentaciji, konkretno tule -
http://docs.python.org/library/re.html - ampak ne med izpitom. Zapiski o
regularnih izrazih so na http://ucilnica.fri.uni-
lj.si/mod/resource/view.php?id=5482 (deluje pa tudi
https://ucilnica.fri.uni-lj.si/mod/resource/view.php?id=5482)."""
>>>
>>> najdi_URLje(s)
['http://www.python.org', 'http://docs.python.org/library/re.html',
'http://ucilnica.fri.uni-lj.si/mod/resource/view.php?id=5482',
'https://ucilnica.fri.uni-lj.si/mod/resource/view.php?id=5482).']
```

Naj te ne vznemirja, da bodo nekateri URLji (zadnji iz primera, recimo) nekoliko napačni, ker se jih bo na koncu držalo še kaj, kar ne sodi zraven.

234. Deli URLja

Lotimo se nekoliko splošnejših URLjev: sestavljeni so iz imena protokola, naslova strežnika in poti na strežniku. Ime protokola je beseda sestavljena iz črk in števk, ki ji sledi dvopičje (http:, https:, ftp: ali kaj podobnega). Imenu protokola sledita dve poševnici. V naslovu strežnika ni poševnic, v poti pa so lahko. Pot se, če ni prazna, začne s poševnico. Celoten URL je torej tak: protokol://strežnik/pot ali protokol://strežnik/ ali protokol://strežnik.

Napiši funkcijo, ki prejme URL in kot rezultat vrne ime protokola, naslov strežnika in pot. Predpostaviš lahko, da URLju sledi beli prostor.

```
>>> razbijURL("http://ucilnica.fri.uni-lj.si/p1")
('http', 'ucilnica.fri.uni-lj.si', 'p1')
>>> razbijURL("http://ucilnica.fri.uni-lj.si/p1/view.php&id=13")
('http', 'ucilnica.fri.uni-lj.si', 'p1/view.php&id=13')
>>> razbijURL("http://ucilnica.fri.uni-lj.si/")
('http', 'ucilnica.fri.uni-lj.si', '')
>>> razbijURL("http://ucilnica.fri.uni-lj.si")
('http', 'ucilnica.fri.uni-lj.si', '')
```

235. Trgovinski računi

Trgovina izdaja račune takšne oblike:

```
Trgovina z mešanim blagom Stane
Pot k Stanetu 18
4321 Lem

Sepuljke            15.12
Sepuljke (bio)      22.00
posebni popust
Knjiga              13.50
------------------------
skupaj              50.62

Hvala za nakup!
Se priporočamo.
```

Med imenom izdelka (oz. besedo "skupaj") in ceno (oz. skupno vsoto) je tabulator (`\t`). Tabulatorjev na drugih mestih v dokumentu ni. Pred vsoto vedno piše "skupaj". Vsota sledi seznamu kupljenih izdelkov in cen. Besedilo pred seznamom izdelkov in cen ter za njim je lahko poljubno in poljubno dolgo. Prav tako se lahko znotraj seznama izdelkov pojavljajo dodatne vrstice s poljubnim besedilom, kot je na primer gornji "posebni popust".

Sestavite funkcijo `preveri_racun(ime_dat)`, ki kot argument prejme ime datoteke s trgovinskim računom. Funkcija prebere račun in vrne vrednost `True`, če je vsota na računu pravilna in `False`, če ni pravilna ali pa račun ne vsebuje besede "skupaj" in vsote.

Za gornji račun funkcija vrne `True`, saj je 15.12 + 22.00 + 13.50 res enako 50.62.

236. Izračun računa

Datoteka s cenami vsebuje kode izdelkov, ki se prodajajo v trgovini, in njihove cene na enoto (kos, kilogram, liter, tona, kubik, kontejner, vlačilec s priklopnikom...). Vsaka vrstica se nanaša na en izdelek. Videti je lahko, recimo, takole:

```
2848802640  10.27
5187117715  13.93
2877291451  19.55
5132905819  13.44
2880387257  12.06
2850174450  0.75
5135732020  2.33
2886988508  6.28
```

V drugi datoteki je seznam reči, ki jih je kupila določena oseba; v vsaki vrstici je koda izdelka in količina. Videti je lahko, na primer, takole:

```
2880387257   3
2886988508 1.5
```

Napiši funkcijo `racunaj(cene, nakupi)`, ki kot argument prejme imeni gornjih datotek in kot rezultat vrne skupno ceno vsega nakupljenega. V gornjem primeru mora funkcija vrniti 45.6 saj je 3×12.06 + 1.5×6.28 = 45.6.

237. Numerologija

Numerologi računajo takole. Naj bo vrednost znaka A enaka 1, vrednost znaka B naj bo 2, vrednost C 3, vrednost D 4 in tako lepo naprej po angleški abecedi. Vzemimo neko besedo, denimo "Benjamin". Vrednosti črk te besede so 2, 5, 14, 10, 1, 13, 9 in 14. Če jih seštejemo, dobimo 68. Nato seštejemo 6 in 8: dobimo 14. Nato seštejemo 1 in 4 ter dobimo 5. Ker smo tako prišli do enomestnega števila, se ustavimo: vrednost imena BENJAMIN je 5. Za vajo poračunajmo še Berto: vrednosti črk so 2, 5, 18, 20, 1. Vsota je 46. 4+6=10. 1+0=1. Tu se ustavimo, vrednost imena Berta je 1.

Od te številke je domnevno odvisno prav vse v vašem življenju, vključno s tem, ali boste uspešno opravili izpit iz programiranja ali ne. Zato se zdi, da bi bilo koristno, če bi napisali funkcijo `numerologija(ime)`, ki kot argument dobi ime (same črke, brez presledkov ali drugih znakov) in vrne njegovo vrednost. Male in velike črke imajo iste vrednosti.

```
>>> numerologija("Berta")
1
>>> numerologija("Benjamin")
5
>>> numerologija("Berta") == numerologija("Benjamin")
False
```

Namig: če funkciji `ord` podamo znak, vrne kodo ASCII tega znaka. Kode ASCII velikih črk so za 64 večje od vrednosti črk, kakršne potrebujemo pri tej nalogi: `ord("A")` je 65, `ord("B")` je 66 in tako naprej.

238. Grep

Napiši funkcijo `grep`, ki kot argument dobi opis imena datoteke (npr. `*.txt` ali `*.*` ali `c:\d\fakulteta\mmm*.py`) in podniz, kot rezultat pa vrne seznam datotek, ki vsebujejo ta podniz. Tako bi `grep("c:\\d\\risanje*.py", "zelva")` vrnil imena vseh datotek v direktoriju c:\d\risanje in s končnico *.py, ki vsebujejo besedo zelva

```
>>> grep("c:\\d\\risanje\\*.py", "zelva")
['c:\\d\\risanje\\koch.py', 'c:\\d\\risanje\\neobjektno.py',
'c:\\d\\risanje\\objektno.py', 'c:\\d\\risanje\\tavajoci.py',
'c:\\d\\risanje\\tavajoci2.py', 'c:\\d\\risanje\\tavajoci3.py',
'c:\\d\\risanje\\tavajoci4.py']
```

Pomagaš si lahko s funkcijo `glob`, ki jo najdeš v modulu `glob`. Kot argument ji podaš ime datoteke (z zvezdicami) in vrne seznam vseh datotek, ki ustrezajo vzorcu.

```
>>> glob.glob("c:\\d\\risanje\\*.py")
['c:\\d\\risanje\\besede.py', 'c:\\d\\risanje\\crte.py',
'c:\\d\\risanje\\koch.py', 'c:\\d\\risanje\\sierpinski.py']
```

239. Preimenovanje datotek

Sestavite program, ki popravi imena datotek v trenutnem direktoriju v skladu z naslednjimi pravili.

1. Spreminjamo samo datoteke s končnicami .avi, .mpg, .mkv, .rm in .mp4.
2. Vse pike v imenu je potrebno spremeniti v presledke. Izjema je pika pred končnico. Tako se mora "generals.at.war-the.battle.at.kursk.avi" spremeniti v "generals at war - the battle at kursk.avi"
3. Prva beseda mora biti vedno napisana z veliko začetnico ("Generals at war - the battle at kursk.avi")
4. Če se v besedilu pojavi pomišljaj, mora biti beseda za njim napisana z veliko začetnico ("Generals at war - The battle at kursk.avi"). Pomišljaj je znak "-", pred in za katerim je presledek. Tule ni pomišljaja: "newton-leibniz".
5. Z veliko začetnico morajo biti napisane tudi vse ostale besede, razen končnice in besed a, an, the, above, against, along, alongside, amid, amidst, around, as, aside, astride, at, athwart, atop, before, behind, below, beneath, beside, besides, between, beyond, but, by, down, during, for, from, in, inside, into, of, on, onto, out, outside, over, per, plus, than, through, throughout, till, to, toward, towards, under, underneath, until, upon, versus, via, with, within, without. Zaradi enostavnosti predpostavljamo, da je beseda od ostalega besedila ločena s presledki. To pomeni, da v naslovu "More on Newton-leibniz Discussion.avi" besede leibniz ne obravnavamo kot besedo (ker pred njo ni presledka).
6. Končnica mora biti napisana z malimi tiskanimi črkami.
7. Zaporedne presledke smemo zamenjati z enim samim presledkom.

original	popravljeno ime
by.the.lake.mpg	By the Lake.mpg
generals.at.war.-.the.battle.at.kursk.avi	Generals at War - The Battle at Kursk.avi
Hubble, 15 Years Of Discovery.avi	Hubble, 15 Years of Discovery.avi
Inside the Forbidden City - 1 - secrets.AVI	Inside the Forbidden City - 1 - Secrets.avi
Killer.toads.avi	Killer Toads.avi
Ne.Preimenuj.Me!.txt	Ne.Preimenuj.Me!.txt
README	README

240. Vse datoteke s končnico .py

Napišite funkcijo, ki prejme ime direktorija in vrne seznam vseh datotek v tem direktoriju, ki imajo končnico .py.

To je bilo seveda le za ogrevanje (kdo pa ne ve za `os.listdir` ali celo `glob`?!). Zdaj pa še zares: seznam naj vsebuje tudi vse datoteke v poddirektorijih; v seznamu naj bodo imena poddirektorijev in datotek, npr. "prog1/risanje/snezinke.py".

Opomba: naloga je vaja iz rekurzije. Kdor je ne zna in se je nima namena (takoj) učiti, niti se ji ne zna izogniti (kar je še težje), naj se je ne loti.

241. Uredi CSV

Napiši funkcijo, ki kot argument prejme ime datoteke v formatu CSV (comma separated values). Napisati mora novo datoteko, katere ime je enako staremu, le da je k imenu pripeto "`_sorted`"; v novi datoteki morajo biti polja v vsaki vrstici urejena po abecednem vrstnem redu.

Program naj datoteko `foo.csv` z vsebino

```
jabolko,banana
kamela,pingvin,veverica,meerkat
glenda,tux,daemon
```

spremeni v datoteko `foo_sorted.csv` z vsebino:

```
banana,jabolko
kamela,meerkat,pingvin,veverica
daemon,glenda,tux
```

242. Zaporedni samoglasniki in soglasniki

V slovenščini ni veliko besed, ki bi imele po dva zaporedna samoglasnika (a preudarno, govoriti o njihovem neobstoju bi bilo preuranjeno!), prav tako so redke takšne, ki imajo zapored po štiri ali več soglasnikov (književna zvrst gor ali dol). Napiši funkcijo zaporedne(s), ki v podanem nizu s poišče vse takšne besede in jih vrne v seznamu.

243. Migracije

Napiši funkcijo migracije(ime_datoteke), ki prejme ime datoteke, ki vsebuje vrstice oblike

```
8: Maribor -> Ljubljana
3: Maribor -> Nova Gorica
10: Ljubljana -> Maribor
5: Koper -> Nova Gorica
3: Novo mesto -> Nova Gorica
```

Vsaka vrstica pove, koliko ljudi se je preselilo odkod kam. Funkcija naj vrne par imen krajev: kraj, iz katerega je odšlo največ ljudi in kraj, v katerega se je preselilo največ ljudi. V gornjem primeru funkcija vrne `("Maribor", "Nova Gorica")`.

244. Selitve

Napiši funkcijo `selitve(zacetek, datoteka_selitev, kraj)`, ki prejme začetno razdelitev oseb, ime datoteke s selitvami in ime nekega kraja. Vrniti mora množico imen oseb, ki po podanih selitvah živijo v podanem kraju.

Začetna razdelitev oseb po krajih je lahko, recimo, `[("Ljubljana", "Jana"), ("Šentvid", "Vid"), ("Mala Polana", "Ana"), ("Maribor", "Bor"), ("Kamnik", "Nik"), ("Ozeljan", "Jan"), ("Županje Njive", "Ive"), ("Koroška Bela", "Ela"))]`. Vedno le po ena oseba v vsakem mestu.

Argument datoteka_selitev pove ime datoteke z vsebino, ki je lahko, recimo, taka

```
Najprej grejo iz "Mala Polana" v "Šentvid".
Iz "Kamnik" pa tudi v "Šentvid".
Pa še iz "Koroška Bela" odidejo v "Ozeljan".
Potem pa iz "Šentvid" v "Maribor" (kwa?!).
```

Vsaka vrstica torej vsebuje dve imeni krajev, zapisani v dvojnih narekovajih. V vrstici ni drugih dvojnih narekovajev. Vedno se preselijo vsi prebivalci podanega kraja.

Če je gornje začetno stanje shranjeno v seznamu zacetek, one štiri vrstice pa v datoteki selitve.txt, mora klic `selitve(zacetek, "selitve.txt", "Maribor")` vrniti `{"Ana", "Vid", Nik", "Bor"}`, klic `selitve(zacetek, "selitve.txt", "Šentvid")` pa vrne prazno množico (saj so šli Šentvidčani v Maribor).

245. Navodila

Napiši funkcijo `navodila(zaporedje)`, ki prejme zaporedje, kot je, na primer `"7D3221LL"` in vrne seznam `[7, "D", 3221, "L", "L"]`. Vsako število spremeni v število (lahko tudi večmestno), črke L in D, ki se poleg tega pojavljajo v nizu, pa vključi kot posamične črke.

246. Poet tvoj nov Slovencem venec vije

Predpostavimo, da se dve besedi rimata, če se ujemata v zadnjem zlogu. Napiši funkcijo, ki prejme kitico pesmi v obliki večvrstičnega niza: vsaka vrstica pesmi je ena vrstica niza. Besede

so ločene s presledki, v besedilu so tudi vejice in druga ločila. Funkcija naj vrne niz, ki pove, kakšne oblike je rima. To določimo tako, da

- vrstici, ki se ne rima z nobeno predhodno vrstico, priredimo novo črko; črke jemljemo po abecedi
- vrstici, ki se rima s predhodno, priredimo enako črko kot tej vrstici.

Spodaj je nekaj primerov.

Po-ét tvoj nòv Slo-ven-cem vê-nec vi-je,
'z pet-nájst so-ne-tov tí ta-kó ga splé-ta,
de »Mà-gi-strá-le«, pe-sem tri-krat pé-ta,
vseh dru-gih sku-paj vé-že hà-rmo-ní-je.

Rima: ABBA

Ti sì živ-ljê-nja moj-'ga mà-gi-strá-le,
gla-síl se 'z njê-ga, kò ne bó več mê-ne,
ran mô-jih bò spo-min in tvô-je hva-le.

Rima: ABA

To ni pe-sem,
so le ne-ke pi-sa-ri-je,
kar brez ri-me,
(če se kam ne skri-je).

Rima: ABCB

Rešitve

Čisti začetek

1. Pretvarjanje iz Fahrenheitov v Celzije

Rešitev zahteva, da znamo uporabnika kaj vprašati in da znamo kaj izpisati.

```
F = float(input("Temperatura [F]: "))
C = 5 / 9 * (F - 32)
print("Temperatura je", C, "C")
```

Začetnik, ki mu je naloga tudi namenjena, bo opazil dva `C`ja v `print`u, eden je pod narekovaji, eden ne. Oni brez narekovajev zahteva izpis *spremenljivke* C, oni z njimi izpis *niza* C (torej, črke C). Prav tako je pod narekovaji drugo besedilo, ki ga je potrebno izpisati takšnega, kot je.

Tudi prva vrstica utegne koga zmesti. Funkcija `input` vrne niz, besedilo, ki ga je vpisal uporabnik. Preden lahko karkoli pametnega naredimo z njim, ga moramo s funkcijo `float` pretvoriti v število. Če je to za koga še preveč, naj celo vrstico jemlje kot čarobne besede, katerih rezultat je, da računalnik nekaj vpraša uporabnika in vrne število, ki ga je ta vpisal. Bo že kmalu razumel njihov pomen.

Program v drugo smer je enak, le z drugo formulo.

```
C = float(input("Temperatura [C]: "))
F = 9 / 5 * C + 32
print("Temperatura je", F, "F")
```

Nezačetnik je morda napisal

```
print("Temperatura je", 5 / 9 * (float(input("Temperatura [F]: ")) - 32), "C")
```

Da, frajerji znajo pisati programe v eni vrstici. Pa je to – v tem primeru – kaj posebnega? Prav nič. Rešitve v eni vrstici bomo, tudi v tej zbirki, z veseljem napisali, kadar bodo duhovite, kadar bodo zahtevale posebno iznajdljivost ali pa se bomo ob njih naučili kaj novega. Takšnemu nečitljivemu programiranju, kot je tole, pa se bomo izogibali, prav?

Rešitvi s funkcijami sta nepresenetljivo kratki.

```
def celsius(f):
    return (f - 32) * 5 / 9

def fahrenheit(c):
    return c * 9 / 5 + 32
```

2. Pitagorov izrek

Za začetek moramo poznati `from math import *`, čarobne besede, ki nam omogočijo uporabo matematičnih funkcij. (Nezačetnik ve, da gre za uvoz modula.) Ko jih izrečemo, smemo v programu klicati funkcije, kot so `sin`, `cos`, `sqrt` in podobne.

```
from math import *
a = float(input("Prva kateta: "))
b = float(input("Druga kateta: "))
c = sqrt(a ** 2 + b ** 2)
print("Hipotenuza trikotnika s stranicama", a, "in", b, "je", c)
```

Začetnik bo spet pozoren na izpis: `a`, `b` in `c` so spremenljivke. Funkcija `print` mora izpisati njihove vrednosti. Za razliko od tega so "*Hipotenuza trikotnika s stranicama*", "*in*", in "*je*" besedila, ki bi jih radi izpisali. Ta pa moramo zapreti v narekovaje. Če dodamo narekovaje `a`-ju, smo namesto spremenljivke `a` dobili niz, v katerem je "besedilo" *a*. Program bo izpisal dobesedno to, namreč *a*. Če pa izpustimo narekovaje pri `"je"`, bomo od `print`a zahtevali izpis vrednosti spremenljivke `je`. Takšne spremenljivke ni in program bo javil napako.

Rešitev s funkcijo je

```
def hipotenuza(a, b):
    return sqrt(a ** 2 + b ** 2)
```

3. Topologija

Kot v prejšnjih dveh nalogah: vprašamo–računamo–izpišemo. Težave nam dela samo funkcija `sin`. Zamolčali smo, da zahteva kot v radianih, ne stopinjah. Stvar ni brez pedagoškega smotra: morda ste reč sprogramirali in sami odkrili, da je z vašim programom nekaj narobe, morda pa bo dobil uporabnik vašega programa topovsko kroglo na lastno glavo. Če ste odkrili, da je nekaj narobe, ste, upamo, spustili program do formule, ki izračuna razdaljo, ter secirali, kaj se dogaja v njej. Ko ste odkrili, da je nekaj narobe s sinusom, ste morda pogledali dokumentacijo funkcije. Kdor je šel po tej poti, je ravnal prav. Google je vaš prijatelj, ni pa edini: pogosto se izplača tudi branje dokumentacije. Kdor je guglal z *error in sine function*, ni prišel daleč; kar piše v (kratki) dokumentaciji funkcije `sin`, pa bi moralo zadoščati.

```
>>> from math import *
>>> help(sin)
sin(...)
    sin(x)

Return the sine of x (measured in radians).
```

Res, ko neka funkcija ne dela, poglej najprej dokumentacijo funkcije, šele potem išči druge, ki so reševali isti problem.

Funkcija za izračun sinusa (prav tako pa tudi vse druge kotne funkcija) v vseh normalnih programskih jezikih sprejema kote v radianih. Stopinje so za ljudi; računalniki, fiziki in

matematiki pa vedo le za radiane (fizik se pri angleškem valčku obrača po $\pi/2$, pri dunajskem pa za cel π in z malo večjo kotno hitrostjo).

```
from math import *
g = 10
kot = float(input("Vnesi kot (v stopinjah): "))
v = float(input("Vnesi hitrost (v m/s): "))
kot_rad = kot * 2 * pi / 360
razdalja = v ** 2 * sin(2 * kot_rad) / g
print("Kroglo bo odneslo", razdalja, "metrov.")
```

Za pretvarjanje iz stopinj v radiane obstaja tudi funkcija: namesto

```
kot_rad = kot * 2 * pi / 360
```

bi lahko pisali

```
kot_rad = radians(kot)
```

Rešitev s funkcijo pa je

```
def top(v, fi):
    return v ** 2 * sin(2 * radians(fi)) / 10.81
```

4. Ploščina trikotnika

Brez posebnega preverjanja bi bil program takšen.

```
from math import *
a = float(input("Prva stranica: "))
b = float(input("Druga stranica: "))
c = float(input("Tretja stranica: "))
s = (a + b + c) / 2
p = sqrt(s * (s - a) * (s - b) * (s - c))
print("Ploščina trikotnika je", p)
```

Če kot dolžine stranic vpišemo 1, 2 in 5, se Python pritoži, da ne more koreniti negativnih števil. Res, s je v tem primeru (1+2+5)/2 = 4 in $s(s-a)(s-b)(s-c) = 4 \times 3 \times 2 \times (-1) = -24$. Nepravilni trikotniki so takšni, v katerih je dolžina ene od stranic daljša od vsote ostalih dveh. Vendar ne bomo pisali

```
if a > b + c or b > a + c or c > a + b:
    print("Nepravilen trikotnik")
```

Ni potrebno. Raje bomo kar izračunali tisto, kar je pod korenom, in preverili, ali je pozitivno. Negativno bo natanko takrat, ko je trikotnik "ilegalen".

```
from math import *
a = float(input("Prva stranica: "))
b = float(input("Druga stranica: "))
c = float(input("Tretja stranica: "))
s = (a + b + c)/ 2
p2 = s * (s - a) * (s - b) * (s - c)
if p2 < 0:
    print("Takšnega trikotnika vendar ni!")
else:
    p = sqrt(p2)
    print("Ploščina trikotnika je", p)
```

Produkt pod korenom smo, zavoljo preglednosti, označili s `p2` (napisali bi p^2, pa Python tega ne pusti.)

Tule pa je še funkcija.

```
def ploscina_trikotnika(a, b, c):
    s = (a + b + c) / 2
    return sqrt(s * (s - a) * (s - b) * (s - c))
```

Pogoji in zanke

5. Vaja iz poštevanke

Ko sta števili izžrebani – kako, smo videli v navodilih iz naloge, izpišemo račun, napišemo vprašanje (»Odgovor?«), nato pa preverimo, ali je uporabnikov odgovor enak produktu.

```
from random import *

a = randint(2, 10)
b = randint(2, 10)
print(a, "krat", b)
c = int(input("Odgovor? "))

if a * b == c:
    print("Pravilno.")
else:
    print("Napačno.")
```

Popaziti moramo na različne številske tipe: koti, pod katerimi streljamo iz topa in dolžine stranic trikotnikov, zato smo uporabnikov vnos pretvarjali v število s funkcijo `float`. Tale produkt pa je celo število, zato smo ga namesto s `float` pretvorili z `int`.

6. Vsi po pet

Naloge se lahko lotimo z zanko `while`.

```
vsota = 0
izdelek = 0
while izdelek < 5:
    cena = float(input('Cena izdelka: '))
    vsota += cena
    izdelek += 1
print('Vsota:', vsota)
```

Spremenljivka `izdelek` šteje, koliko reči imamo v košarici. Šla bo od 0 do 4. V vsakem krogu uporabnika vprašamo po ceni izdelka in jo prištejemo (`+=`) k vsoti. Pisali bi lahko tudi `vsota = vsota + cena`, a ne bomo; `vsota += cena` je krajše in običajnejše. Podobno je z `izdelek += 1`, ki je običajna bližnica za `izdelek = izdelek + 1`.

Začetnik se ob teh nalogah prvič srečuje s pogostim vzorcem: seštevanje v zanki. To bomo videli še velikokrat, le da bomo enkrat prištevali števila, drugič bomo dopisovali besedilo v niz, spet tretjič dodajali elemente v seznam.

Kdor se še ni lotil nalog s seznami, naj ne bere naprej. Kdor se je, pa že sme vedeti: preprosteje kot z `while` rešimo nalogo s `for`. Zakaj? Ker zanki `for` naročimo, naj nekaj naredi petkrat. Ko

smo uporabili `while`, smo morali šteti sami – `izdelek` smo morali sami postaviti na `0`, ga sami povečevati za `1` in preverjati, ali je že dosegel `5`. Zanka `for` to počne sama.

```
vsota = 0
for izdelek in range(5):
    cena = float(input('Cena izdelka: '))
    vsota += cena
print('Vsota:', vsota)
```

7. Konkurenca

Rešitev je podobna rešitvi naloge Vsi po pet. Tudi te se lahko lotimo z zanko `while`

```
izdelkov = int(input('Stevilo izdelkov: '))
izdelek = 0
vsota = 0
while izdelek < izdelkov:
    cena = float(input('Cena izdelka: '))
    vsota += cena
    izdelek += 1
print('Vsota:', vsota)
```

Navadno bomo namesto `while` raje uporabili `for` (spet samo za tiste, ki že poznajo zanko `for` prek številskih intervalov):

```
izdelkov = int(input('Stevilo izdelkov: '))
vsota = 0
for izdelek in range(izdelkov):
    cena = float(input('Cena izdelka: '))
    vsota += cena
print('Vsota:', vsota)
```

Razlika je le v tem, da smo prej puščali zanko, da je tekla do 5, zdaj pa v začetku povprašamo po številu izdelkov in zanko ponovimo, kolikorkrat je potrebno.

Spet smo pazili na različne številske tipe: cene izdelkov v trgovinah so necela števila, zato smo uporabnikov vnos pretvarjali v število s funkcijo `float`. Število izdelkov je celo (v košarici imamo 5 ali 6 reči, nikoli pa 5,28), zato smo ob branju števila izdelkov namesto `float` uporabili `int`.

8. Top-shop

Za reševanje je najprimernejša zanka `while`.

Tistim, ki že poznajo `for`, povejmo, zakaj za tole ni primerna. Zanko `for` namreč navadno uporabimo, kadar vnaprej vemo, kolikokrat jo želimo ponoviti, tule pa bo moral program končati, ko je izpolnjen določen pogoj, namreč, ko bo vpisana cena enaka 0. Če bi uporabili zanko `for`, ne bi vedeli niti, kako postaviti mejo. Bomo rekli, da noben kupec ne bo kupil več kot milijon izdelkov? Tedaj lahko začnemo z

```
for izdelek in range(1000000):
    cena = float(input('Cena izdelka: '))
    if cena == 0:
        break
```

in tako naprej. Vendar to ni videti prav lepo, tej nalogi je na kožo pisan `while`.

```
vsota = 0
cena = 1
while cena != 0:
    cena = float(input('Cena izdelka: '))
    vsota += cena
print('Vsota:', vsota)
```

Pozoren bralec je opazil, da prištejemo k vsoti tudi izdelek s ceno 0. 0 ne de.

9. Državna agencija za varstvo potrošnikov

V program je potrebno dodati spremenljivko, ki šteje izdelke. V začetku jo bomo postavili na 0 in jo povečali ob vsakem novem izdelku. Po zanki jo zmanjšamo za 1, da ne štejemo zadnje vpisane cene, 0. Poprečno ceno izpišemo le, če je uporabnik kupil vsaj en izdelek.

```
izdelek = 0
vsota = 0
cena = 1
while cena != 0:
    cena = float(input('Cena izdelka: '))
    vsota += cena
    izdelek += 1
izdelek -= 1

print('Vsota:', vsota)
if izdelek > 0:
    print('Poprecna cena:', vsota / izdelkov)
```

V začetku smo ceno postavili na 1. To je potrebno in neškodljivo. Potrebno, ker zanka `while` na začetku preverja, ali je cena različna od 0; spremenljivka `cena` torej mora obstajati in biti mora različna od 0. Neškodljivo, ker na začetku zanke uporabnik tako ali tako vnese pravo ceno prvega izdelka.

Program bi se dalo skrajšati tako, da bi `izdelek` v začetku postavili na `-1` namesto na `0`, pa nam ga po zanki ne bi bilo potrebno zmanjšati. Profi bi verjetno ubral to, pa še kako drugo bližnjico.

V tej nalogi pravzaprav tudi `for` ne bi izgledal tako smešno kot v prejšnji, saj nas reši preštevanja izdelkov. Da ga uporabimo, pa moramo poznati `break`.

```
vsota = 0
for izdelek in range(1000000):
    cena = float(input('Cena izdelka: '))
    if cena == 0:
        break
    vsota += cena

print('Vsota:', vsota)
if izdelek > 0:
    print('Poprecna cena:', vsota / izdelek)
```

Še število izdelkov je zdaj po zanki ravno pravo. Kar premislimo: če nekdo kupi eno samo reč, bo imela spremenljivka `izdelek` v prvem krogu zanke vrednost 0; blagajnik bo vpisal ceno prvega izdelka. V naslednjem krogu bo imela `izdelek` vrednost 1; blagajnik bo vpisal 0. Zanka se konča in spremenljivka `izdelek` je 1, natanko toliko, kot mora biti. Če bi stranka kupila kaj več, bi bil števec `izdelek` pač ustrezno večji.

10. Collatzova domneva

Naloga ima dve drobni pasti.

```
n = int(input("Vnesi število: "))
while n != 1:
    print(n)
    if n % 2 == 0:
        n /= 2
    else:
        n = n * 3 + 1
print(1)
```

Tale program ne izpiše zadnje številke. Lahko ga obrnemo tako, da postavimo `print` na konec.

```
n = int(input("Vnesi število: "))
while n != 1:
    if n % 2 == 0:
        n /= 2
    else:
        n = n * 3 + 1
    print(n)
```

Nič ne pomaga, zdaj manjka prva številka. Zanka se pač obrne tolikokrat, kolikor korakov deljenja oziroma množenja naredimo, izpisati pa moramo eno število več, kot je korakov. En `print` bo pred zanko ali po njej. Postavimo ga za njo. Napišimo kar `print(1)`, saj vemo, da bo zadnji člen gotovo 1.

```
n = int(input("Vnesi število: "))
while n != 1:
    print(n)
    if n % 2 == 0:
        n /= 2
    else:
        n = n * 3 + 1
print(1)
```

Različica, s katero bi se lahko znebili dvojnega izpisovanja, je takšna:

```
n = int(input("Vnesi število: "))
while True:
    print(n)
    if n == 1:
        break
    if n % 2 == 0:
        n /= 2
    else:
        n = n * 3 + 1
```

Zanka zdaj steče enkrat več, saj jo, ko pridemo do enice, prekine šele `break`. Vendar je prva rešitev, z dvojnim izpisom, elegantnejša, saj nam glava zanke (`while n != 1`) že pove, kdaj se zanka konča. V drugi različici nam glava (`while True`) pove le, da bomo nekje znotraj zanke našli nek `break`, ki jo prekine. Grdo, grdo. Imamo pa že raje dodatni `print` kot tole.

Še ena neželena reč: v izpisu se pojavljajo nesmiselne decimalke – namesto 4, recimo, se izpiše 4.0. Problem je v deljenju. Če bomo uporabljamo običajno deljenje in pišemo `n /= 2`, je rezultat necelo število, kar Python pove tako, da izpiše vsaj eno decimalko (četudi je ta 0). Uporabiti moramo celoštevilsko deljenje, `n //= 2`.

```
n = int(input("Vnesi število: "))
while n != 1:
    print(n)
    if n % 2 == 0:
        n //= 2
    else:
        n = n * 3 + 1
print(1)
```

11. Benjaminovi kovanci

Kako lep primer za preprosto zanko `while`! Dokler je število kovancev med 0 in 10, vržemo kovanec. Če pade grb, se število kovancev zmanjša za 1, sicer poveča.

```
kovancev = 5
while 0 < kovancev < 10:
    met = vrzi()
    if met == "G":
        kovancev -= 1
    else:
        kovancev += 1
    print(met, kovancev)
```

Ne spreglejte, kako elegantno je napisan pogoj: `0 < kovancev < 10`. Šlo bi tudi `kovancev > 0 and kovancev < 10` ali kaj podobnega, a tole je veliko krajše, berljivejše, naravnejše.

Nekateri študenti, ki so to nalogo dobili za domačo nalogo, so pisali

```
kovancev = 5
while True:
    met = vrzi()
    if met == "G":
        kovancev -= 1
    else:
        kovancev += 1
    print(met, kovancev)
    if kovancev == 0 or kovancev == 10:
        break
```

To deluje, a tako se ne programira. Sem ter tja bomo res potrebovali `while True` in tudi k `break`u se bomo pogosto zatekli. Takole pa ne. V tej zanki smo naravnost *skrili*, kar je najpomembnejše, namreč to, kdaj se zanka zaključi.

Naravnost neokusno začetništvo je pisati

```
    if met == "G":
        kovancev -= 1
    elif met == "C":
        kovancev += 1
```

Če je `met` lahko samo G ali C, ne pišemo `elif`, temveč `else`.

Posebna nerodnost pa je

```
kovancev = 5
while 0 < kovancev < 10:
    if vrzi() == "G":
        kovancev -= 1
    else:
        kovancev += 1
    print(vrzi(), kovancev)
```

ali celo

```
kovancev = 5
while 0 < kovancev < 10:
    if vrzi() == "G":
        kovancev -= 1
    elif vrzi() == "C":
        kovancev += 1
    print(vrzi(), kovancev)
```

Funkcija `vrzi` vsakič meče na novo. Tule v vsakem krogu zanke mečemo trikrat. Program ne deluje pravilno, kar takoj vidimo, če ga poženemo.

12. Tekmovanje iz poštevanke

Rešitev ne zahteva posebnih trikov, le pridno programiranje.

```
prvi = drugi = 0

while abs(prvi - drugi) < 2:
    f1 = int(input("Tekmovalec 1, prvi faktor? "))
    f2 = int(input("Tekmovalec 1, drugi faktor? "))
    pr = int(input("Tekmovalec 2, produkt? "))
    print()
    if pr == f1*f2:
        drugi += 1

    f1 = int(input("Tekmovalec 2, prvi faktor? "))
    f2 = int(input("Tekmovalec 2, drugi faktor? "))
    pr = int(input("Tekmovalec 1, produkt? "))
    print()
    if pr == f1*f2:
        prvi += 1

    print("Trenutni rezultat: ", prvi, ":", drugi)
    print()

if prvi > drugi:
    print("Bravo, prvi! Drugi, cvek!")
else:
    print("Bravo, drugi! Prvi, cvek!")
```

Ko so ta problem za domačo nalogo reševali študenti, jih je žalostno malo uporabilo funkcijo `abs`. Raje so pisali

```
while prvi - drugi < 2 and drugi – prvi < 2:
```

Ni lepo, lahko pa se ob tem spomnimo vsaj, da je mogoče primerjave nizati, in napišemo

```
while -2 < prvi - drugi < 2:
```

13. Števke

Navodila naloge so bila več kot jasna.

```
n = int(input("Vnesi število: "))
while n > 0:
    print(n % 10)
    n //= 10
```

Paziti moramo, da uporabimo celoštevilsko deljenje, `//=` in ne `/=`, sicer se reč ne bo končala ne dobro ne hitro. Več o tem v naslednji nalogi, ki kaže isto mehaniko naprej in nazaj.

Funkcija je takšna.

```
def brez_sedmic(n):
    while n > 0:
        print(n % 10)
        n //= 10
```

14. Obrnjena števila

Rešitev zahteva nekaj preproste matematike: vpisanemu številu pobiramo števke od zadaj – podobno kot v prejšnji nalogi – in jih od zadaj potiskamo v novo število. Zadnjo števko dobimo tako, da izračunamo ostanek po deljenju z 10; po tem je potrebno število še deliti z 10, da to števko "odbijemo". Za deljenje spet ne uporabljamo operatorja `/`, temveč `//`, saj želimo celoštevilsko deljenje: `8732 // 10` je `873`; če bi računali `8732/10`, bi dobili `873.2`. Števko porinemo v novo število tako, da novo število pomnožimo z `10` (tako naredimo "prostor" za novo števko) in prištejemo novo števko. Vse skupaj ponavljamo, dokler (`while`) je `n` večji od `0`.

```
n = int(input('Vpisi število: '))
m = 0
while n > 0:
    zadnja_stevka = n % 10
    m = m * 10 + zadnja_stevka
    n = n // 10
print(m)
```

Program je zaradi razumljivosti napisan malo daljše. V resnici nihče ne bi delal tako, raje bi napisali

```
n = int(input('Vpisi število: '))
m = 0
while n > 0:
    m = m * 10 + n % 10
    n //= 10
print(m)
```

V funkciji je to takole:

```
def obrni_stevilo(n):
    m = 0
    while n > 0:
        m = m * 10 + n % 10
        n //= 10
    return m
```

Namesto da bi število `n` prebrali, ga dobimo kot argument. Namesto da bi ga izpisali, ga vrnemo.

Spremenljivka `zadnja_stevka` je bila nepotrebna, prenesli smo jo v izračun m-ja. Prav tako smo "*n-ju priredi n deljen z 10*" (`n = n // 10`) spremenili v "*n deli z 10*" (`n //= 10`).

Enako kot tule bi nalogo rešili v Cju ali kakem drugem jeziku. Skriptni jeziki pa so bolj prilagojeni delu z besedili in nam, kot smo spoznali pri poštevanki števila 7, navadno omogočajo tudi kako bližnjico. V Pythonu lahko rečemo kar

```
n = input('Vpisi število: ')
print(n[::-1])
```

Tule je `n` ostal niz (poklicali smo le `input`, `int` pa izpustili), zato ga lahko obrnemo kot niz. Če hočemo dobiti obrnjeno število in ne niza z obrnjenim številom, lahko obrnjeni niz pretvorimo v število z `int(n[::-1])`.

Če pišemo funkcijo, ta trik seveda ne vžge, saj bomo dobili število, ne niza. Nič ne de: število pretvorimo v niz, ga obrnemo in pretvorimo nazaj v število.

```
def obrni_stevilo(n):
    return int(str(n)[::-1])
```

15. Kalkulator

Nalogo lahko rešimo s kupom `if`-ov.

```
def calc(v1, v2, oper):
    if oper == '+':
        return v1 + v2
    if oper == '-':
        return v1 - v2
    if oper == '*':
        return v1 * v2
```

Kdor zna programirati v kakem drugem jeziku je ob reševanju morda brskal, kako se v Pythonu napiše stavek `switch`. Ni ga. Bi bil program z njim kaj krajši? Ne bistveno. Sploh pa takšne stvari v Pythonu delamo redko. Vsak programski jezik ima stvari, ki jih potrebujemo za učinkovito pisanje programov, kakršne pišemo v tem programskem jeziku.

Skriptni jeziki imajo navadno funkcijo, ki zna evaluirati izraz ali izvesti del programa, ki je podan kot niz. V Pythonu ji je ime `eval`; podamo ji niz, pa ga izračuna. Nalogo bomo z njim (če znamo poleg tega še oblikovati nize s pomočjo znakov `%`), rešili preprosteje.

```
def calc(v1, v2, oper):
    return eval('%i %s %i' % (v1, oper, v2))
```

Zanke prek seznamov in nizov

16. Vsota elementov seznama

Pa jo sprogramirajmo, klasiko klasike.

```
s = [5, 8, 3, 6, 0, 1]

vsota = 0
for e in s:
    vsota += e
print(vsota)
```

Nekateri podležejo skušnjavi, da napišejo

```
s = [5, 8, 3, 6, 0, 1]

vsota = 0
for e in s:
    if e != 0:
        vsota += e
print(vsota)
```

Po tem ni nobene potrebe. Python zna prišteti tudi nič.

Python ima tudi funkcijo sum.

```
s = [5, 8, 3, 6, 0, 1]

vsota = sum(s)
print(vsota)
```

Ta naloga z uporabo funkcije `sum` izgubi smisel. Zdaj pa, ko bi znali vsoto napisati tudi sami, bomo v prihodnjih nalogah brez zadržkov uporabljali funkcijo `sum`, ki so nam jo pripravili drugi.

Rešitev v obliki funkcije je

```
def vsota(s):
    vsota = 0
    for e in s:
        if e != 0:
            vsota += e
    return vsota
```

Natančno tako kot rešitev brez funkcije, le da `s` dobimo kot argument, `vsota` pa vrnemo kot rezultat.

17. Ajavost nizov

Zanko `for` pošljemo prek niza. Vsakič, ko v zanki naletimo na znak "a", k števcu a-jev prištejemo 1.

```
niz = input('Vpiši niz: ')

stevec = 0
for crka in niz:
    if crka == 'a':
        stevec += 1
print('Število a-jev:', stevec)
```

Povejmo še, da znajo nizi prešteti, kolikokrat vsebujejo posamezno črko. Kdor to ve, napiše precej krajši program.

```
niz = input('Vpiši niz: ')
print('Število a-jev:', niz.count('a'))
```

Funkcija, ki vrne število a-jev, je

```
def stevilo_ajev(beseda):
    stevec = 0
    for crka in beseda:
        if crka == 'a':
            stevec += 1
    return stevec
```

Funkcija, ki šteje poljubne znake, ima dodaten argument, `znak`, katerega pojavitve preštevamo. Poleg tega v pogoju ne primerja znaka s (konstantno) črko `"a"` temveč s podanim `znak`om.

```
def stevilo_znakov(beseda, znak):
    stevec = 0
    for crka in beseda:
        if crka == znak:
            stevec += 1
    return stevec
```

Zadnja vaja je zelo pomembna. Najpomembnejša. Naučiti se v svojih funkcijah klicati svoje funkcije, namesto, da eno in isto reč pišemo večkrat. Ker je funkcija za preštevanje znakov splošnejša od fukcije za preštevanje a-jev, bomo a-je prešteli tako, da preštejemo znake a.

```
def stevilo_ajev(beseda):
    return stevilo_znakov(beseda, "a")
```

18. Največji element

Naloga predstavlja eno standardnih fraz v programiranju, ki jo na tisoč in en način ponavljamo v sto in enem kontekstu.

```
s = [5, 1, 4, 8, 2, 3]

najvecji = None
for x in s:
    if najvecji == None or x > najvecji:
        najvecji = x
print(najvecji)
```

V začetku rečemo, da je največji element doslej `None`. Potem pri vsakem preverimo, ali je morda večji od največjega doslej – še pred tem pa, ali je največji doslej morda None (to se seveda zgodi le enkrat, v začetku). V tem primeru si bomo trenutnega, kakršenkoli že je, zapomnili kot največjega.

Tisti, ki že malo boljše poznajo Python in se držijo bontona, namesto `x == None` pišejo `x is None`. Za nas, tule, je to vseeno.

Namesto `None` lahko kot začetno vrednost največjega elementa uporabimo tudi prvi element seznama. To seveda deluje le, če seznam ni prazen.

```
s = [5, 1, 4, 8, 2, 3]

najvecji = s[0]
for x in s:
    if x > najvecji:
        najvecji = x
print(najvecji)
```

Oblika z `None` je splošnejša in nam bo prišla prav tudi v naslednjih nalogah, zato se je bomo držali.

Prišleki iz drugih jezikov bi morda napisali

```
s = [5, 1, 4, 8, 2, 3]

najvecji = s[0]
for i in range(len(s)):
    if s[i] > najvecji:
        najvecji = s[i]
print(najvecji)
```

V Pythonu se to ne šteje za lepo. Kdor bi napisal tako, naj le primerja in razmisli. Kdor na drugo rešitev ni niti pomislil, naj vztraja v tej drži. V tej zbirki bomo naloge, v katerih uporabljamo zanko `for` na ta način, namerno pustili za kasneje.

Še rešitev s funkcijo.

```
def najvecji(s):
    naj = None
    for x in s:
        if naj is None or x > naj:
            naj = x
    return naj
```

Spremenljivko `najvecji` smo preimenovali v `naj`, da se ne tepe z imenom funkcije. Saj bi delovalo, ni pa lepo.

V Python je sicer že vdelana funkcija `max`, ki vrne največji element seznama. Ves program bi se z njo spremenil v

```
s = [5, 1, 4, 8, 2, 3]
print(max(s))
```

Vendar s tem vaja izgubi pomen.

19. Največji absolutist

Če hočemo največje število po absolutni vrednosti, moramo le spremeniti pogoj

```
x > najvecji
```

v

```
abs(x) > abs(najvecji)
```

Tudi tu lahko uporabimo funkcijo `max`, le da moramo znati malo več.

```
xs = [5, -1, 4, -8, 2, 3]
print(max(xs, key=abs))
```

V funkciji je to videti takole

```
def najvecji_abs(s):
    naj = None
    for x in s:
        if naj == None or abs(x) > abs(naj):
            naj = x
    return naj
```

20. Najmanjši pozitivist

Po tem, kar smo počeli v prejšnjih nalogah, je to odlična vaja iz sestavljanja pogojev. Pravilna oblika je tokrat `x > 0 and (naj is None or x < naj)`. S celo funkcijo:

```
def najmanjsi_poz(s):
    naj = None
    for x in s:
        if x > 0 and (naj == None or x < naj):
            naj = x
    return naj
```

Past, v katero bi lahko padli, je `naj == None or 0 < x < naj`. To ne deluje pri seznamu, recimo, `[-1, -2, -3]`. Ker je pri prvem elementu naj enak None, bo pogoj resničen, čeprav je x negativen in torej nesprejemljiv. Pogoj se mora začeti z `x > 0`. Če to ne drži, nas tudi `naj == None` ne zanima.

Druga napaka, ki jo lahko naredimo, je, da pozabimo oklepaje, `x > 0 and naj is None or x < naj`. Ker ima `and` prednost pred `or`, je to potem isto, kot če bi napisali `(x > 0 and naj is None) or x < naj`. To spet ni tisto, kar smo hoteli, saj bo sprejemalo negativne `x` – kar razmislite, kdaj je to resnično.

21. Najdaljša beseda

Variacija na isto temo, le besedilo moramo razbiti na besede (`split`) in potem delati podobno kot z absolutnimi vrednostmi, le namesto `abs` pišemo `len`, saj ne primerjamo števil po absolutni vrednosti temveč nize po dolžini.

Tokrat imamo tudi vabljivo začetno vrednost, `naj = ""`. To nam poenostavi pogoj, saj se nam ni več potrebno ukvarjati s preverjanjem, ali doslej še nismo našli ničesar; `""` je zanesljivo najkrajša možna beseda in vsaka druga, ki jo bomo odkrili, bo daljša.

```
def najdaljsa(s):
    naj = ""
    besede = s.split()
    for x in besede:
        if len(x) > len(naj):
            naj = x
    return naj
```

In tako kot prej lahko nalogo rešimo kar z vdelano funkcijo `max`, če jo znamo uporabljati na nekoliko naprednejši način. Če je ne, pa za zdaj nič hudega.

```
def najdaljsa(s):
    return max(s.split(), key=len)
```

22. Poprečje

Stvar je podobna vsoti, le na prazne sezname moramo paziti.

```
s = [5, 8, 3, 6, 0, 1]

vsota = 0
for e in s:
    if e != 0:
        vsota += e
if not s:
    print(0)
else:
    print(vsota / len(s))
```

Kaj pomeni `if not s`? Prazni seznami so neresnični. Pogoj `if not s` torej sprašuje, ali je seznam morda prazen.

Zanko bi lahko prestavili tudi znotraj `else`, ali pa reč preobrnili na kak drug način. V obliki funkcije bi lahko napisali, recimo

```
def poprecje(s):
    if not s:
        return 0
    vsota = 0
    for e in s:
        if e != 0:
            vsota += e
    return vsota / len(s)
```

Če je seznam prazen, vrnemo 0 in konec. Sicer seštevamo in delimo.

Če se spomnimo, da imamo funkcijo sum, je stvar seveda lažja.

```
def poprecje(s):
    if not s:
        return 0
    return sum(s) / len(s)
```

Obstajajo tudi bolj kriptične rešitve, recimo

```
def poprecje(s):
    return len(s) and sum(s) / len(s)
```

Tole je tipično za Python. Kdor ne razume, a bi rad, naj pobrska, kako v Pythonu dela operator `and`.

23. Poprečje brez skrajnežev

Preden se lotimo reševanja, razmislimo. Na vrat na nos bi lahko namreč reševali nalogo tako, da bi najprej poiskali največji in najmanjši element (z zanko, tako kot smo navajeni), nato pa (z novo zanko) izračunali vsoto vseh elementov, pri čemer bi pri seštevanju preskočili tista dva z največjo in najmanjšo težo.

Ne le, da bi bilo to po nepotrebnem dolgo in zapleteno, temveč bi bilo celo narobe v primeru, da ima enako največjo ali najmanjšo težo več ljudi. Glede na nalogo naj bi izključili le enega.

Boljše je tako: v isti zanki poiščemo najmanjšo in največjo težo ter računamo vsoto. Nato od vsote odštejemo težo najlažjega in najtežjega državljana ter jo delimo z za dva zmanjšano dolžino seznama.

Če ima seznam le dva elementa, pa bo funkcija vrnila nič, saj nima kaj računati.

```
def poprecje_brez(s):
    if len(s) <= 2:
        return 0
    najm = najv = s[0]
    vsota = 0
    for e in s:
        if e < najm:
            najm = e
        elif e > najv:
            najv = e
        vsota += e
    return (vsota - najm - najv) / (len(s) - 2)
```

Če se spomnimo, da ima Python vdelane funkcije min, max in sum, ki poiščejo najmanjši in največji element ter vsoto njegovih elementov, smo lahko veliko učinkovitejši.

```
def poprecje_brez(s):
    if len(s) <= 2:
        return 0
    return (sum(s) - min(s) - max(s)) / (len(s) - 2)
```

Nekateri pa iz gole objestnosti pišejo

```
def poprecje_brez(s):
    return (sum(s) - min(s) - max(s)) / (len(s) - 2) if len(s) > 2 else 0
```

24. Bomboni

Včasih se splača malo razmisliti, preden začnemo tipkati.

Recimo, da imamo štiri otroke in tisti, ki ima največ, ima osem bombonov. Ko jih bodo imeli vsi enako, jih bodo torej imeli 32. Recimo, da jih imajo sedaj 23. Koliko jim jih moramo še razdeliti? 32 - 23 = 9, ne? Če se iz prejšnjih nalog spomnimo, da ima Python funkciji `max` in `sum`, napišemo:

```
def bomboni(s):
    return max(s) * len(s) - sum(s)
```

Kdor rad programira brez potrebe (tako kot vsi študenti, ki so to nalogo reševali na izpitu, razen enega), pa nalogo reši takole

```
def bomboni(s):
    naj = max(s)
    damo = 0
    for i in s:
        damo += (naj - i)
    return damo
```

Najprej pogledamo, koliko bombonov ima ta, ki jih ima največ (recimo, da vemo vsaj za funkcijo `max`), nato pa seštevamo število bombonov, ki jih bo treba dati posameznemu mulcu.

Oklepaji v `damo += (naj - i)` niso čisto potrebni, a smo jih napisali zaradi preglednosti.

25. Vsaj eno liho

Naloga vsebuje kamen spotike mnogih, ki se učijo programiranja.

```
def vsaj_eno_liho(s):
    for e in s:
        if e % 2 == 1:
            return True
    return False
```

Pomembno je tole: če je število liho (torej, če je ostanek, ki ga dobimo po deljenju z 2, enak 1), takoj vrnemo `True`. Sicer pa iščemo naprej.

Neučakani reševalci naloge (ali pa tisti, ki menijo, da mora vsakemu `if`-u slediti tudi `else`, kot starši porednih otrok, ki morajo otroku vedno zagroziti tudi s kaznijo – pospravi igrače, or else!), radi napišejo takšnole napačno rešitev:

```
def vsaj_eno_liho(s):
    for e in s:
        if e % 2 == 1:
            return True
        else:
            return False
```

To je zelo narobe, ker v resnici preveri le prvo število. Če je liho, vrne `True`, sicer vrne `False` – že takoj, po prvem številu, ne da bi pogledali naprej.

Fraza "vsaj eden" je tako pogosta, da jo lahko v Pythonu (in mnogih drugih jezikih) zapišemo tudi precej krajše. Celotno funkcijo lahko zapišemo kot

```
def vsaj_eno_liho(s):
    return any(e % 2 == 1 for e in s)
```

26. Sama liha

Čeprav je tole zelo podobno prejšnji nalogi, bomo reševali malo drugače. Pokazali bomo več različic, nekatere bodo tudi nekoliko zahtevnejše.

Najprej nekoliko nerodnejša rešitev.

```
def sama_liha(s):
    vsa_liha = True
    for e in s:
        if e % 2 == 0:
            vsa_liha = False
    return vsa_liha
```

Obstaja sicer še nerodnejša; preprosti return na koncu bi lahko zamenjali z

```
    if liha == True:
        return True
    else:
        return False
```

Vendar je to preočitna neumnost. Zgornja, manj nerodna rešitev, torej stori tole: v začetku predpostavi, da bodo vsa števila liha (`liha = True`). Čim naleti na kakšno, ki ni, pa si to zapomni (`liha = False`). Spet, ne smemo se zmotiti in `if`-a nadaljevati z "`else: liha = True`": ko smo enkrat naleteli na sodo, nikoli več ne bo res, da so bila vsa števila, ki smo jih videli, liha.

Če je tako, pa lahko hitro uvidimo tudi, da zanke po tem, ko smo naleteli na sodo število, sploh nima več smisla nadaljevati, temveč jo lahko preprosto prekinemo:

```
def sama_liha(s):
    vsa_liha = True
    for e in s:
        if e % 2 == 0:
            vsa_liha = False
            break
    return vsa_liha
```

Ko vidimo to, da pomislimo, da pravzaprav ne potrebujemo `break`a, saj lahko rečemo kar `return`.

```
def sama_liha(s):
    vsa_liha = True
    for e in s:
        if e % 2 == 0:
            return False
    return vsa_liha
```

Zdaj imamo šele traparijo: `vsa_liha` bo vedno `True`, saj ga vendar nihče nikoli ne postavi na `False`. Zadnji `return` torej spremenimo v `return True` in se znebimo spremenljivke `vsa_liha`.

```
def sama_liha(s):
    for e in s:
        if e % 2 == 0:
            return False
    return True
```

To je, če smo natančni, pravzaprav rešitev neke drugačne naloge: namesto *funkcije, ki vrne `True`, če so v seznamu sama liha števila (in `False`, če je med njimi kakšno sodo)*, smo napisali

funkcijo, ki vrne `False`, če imamo kakšno sodo število (in `True`, če ga ni in so vsa števila liha). Ampak to je pravzaprav eno in isto.

V prejšnji – zelo podobni – nalogi smo sprogramirali frazo "vsaj eden", v tej programiramo frazo "vsi". Tako kot ima Python bližnjico za prvo (`any(e % 2 == 1 for e in s)`), jo ima tudi za to:

```
def sama_liha(s):
    return all(e % 2 == 1 for e in s)
```

27. Blagajna

Z zanko gremo čez niz in spremljamo dolžino vrste: vsakič, ko vidimo +, zabeležimo, da je vrsta za 1 daljša, sicer pa je za 1 krajša. Kadar se vrsta podaljša, še preverimo, ali je daljša od najdaljše doslej in si jo, če je tako, zapomnimo. Na koncu vrnemo dolžino najdaljše vrste.

```
def blagajna(s):
    naj_vrsta = vrsta = 0
    for c in s:
        if c == "+":
            vrsta += 1
            if vrsta > naj_vrsta:
                naj_vrsta = vrsta
        else:
            vrsta -= 1
    return naj_vrsta
```

28. Preobremenjeni čolni

Če poznamo funkcijo `sum`, rešimo nalogo tako:

```
def ni_preobremenjenih(tovori, nosilnost):
    for coln in tovori:
        if sum(coln) > nosilnost:
            return False
    return True
```

Če je ne poznamo, potrebujemo zanko znotraj zanke.

```
def ni_preobremenjenih(tovori, nosilnost):
    for coln in tovori:
        vsota = 0
        for tovor in coln:
            vsota += tovor
        if vsota > nosilnost:
            return False
    return True
```

Tam, kjer bi sicer poklicali `sum`, zdaj vstavimo točno tisto, kar smo naprogramirali, ko smo sami programirali izračun vsote. Upam, da vas zanka znotraj zanke ne moti. Pythona ne.

Dva študenta sta na izpitu suvereno napisala:

```
def ni_preobremenjenih(colni, nosilnost):
    return all(sum(coln) <= nosilnost for coln in colni)
```

Zanke prek številskih intervalov

29. Delitelji

Z zanko `for` štejemo od 1 do podanega števila in preverjamo, ali je ostanek po deljenju enak 0.

```
n = int(input('Vpiši število: '))
for i in range(1, n + 1):
    if n % i == 0:
        print(i)
```

V zanki `for` spet pazimo na meje. Podati moramo spodnjo mejo, 1. Če jo izpustimo, se štetje začne z 0 in program bo javil napako, ko bo poskušal izračunati ostanek po deljenju z 0. Gornja meja je `n + 1`, ker Pythonova funkcija `range` deluje tako, da šteje do zgornje meje, vendar brez nje. Tako, kot smo napisali, bomo torej dobili vsa števila od, vključno, `1` do `n`.

Da se ti, ki rešujejo začetniške naloge, čeprav razumejo že malo več, ne bi predolgo dolgočasili, pokažimo program s pomočjo generatorjev. Seznam vseh deliteljev števila `n` lahko dobimo takole

```
[i for i in range(1, n + 1) if n % i == 0]
```

Če bi radi izpisali vse delitelje, jih moramo pretvoriti v nize in zlepiti skupaj. Celo nalogo tako reši ena sama vrstica

```
print("\n".join(str(i) for i in range(1, n + 1) if n % i == 0))
```

30. Praštevilo

Z zanko `for` štejemo od 2 do n in če naletimo na število, ki deli podano število, vrnemo `False`.

```
def prastevilo(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Ker Pythonov range teče le do zgornje meje, brez nje, bo šel `i` samo do (vključno) `n – 1`, kar je natančno to, kar hočemo.

Število `i` je delitelj `n`-ja, če ga deli brez ostanka, torej, če je ostanek po deljenju `n` z `i` enak `0`.

Le na klasično napako moramo paziti. `False` vrnemo, čim odkrijemo delitelja, `True` pa šele po koncu zanke, torej, če med vsemi števili od `2` do `n-1` nismo odkrili nobenega delitelja.

```
# Tole je zelo pogosta napačna rešitev!!!
def prastevilo(n):
    for i in range(2, n):
        if n % i == 0:
            return False
        else:
            return True # NAROBE!
```

Tovrstne naloge je precej preprosteje rešiti v obliki funkcije, kot brez nje. Poskusimo jo sprogramirati, kot bi to storil kdo, ki še ne zna pisati funkcij.

```
n = int(input("Vnesi število: "))
for i in range(2, n):
    if n % i == 0:
        print(n, "ni praštevilo")
        break
else:
    print(n, "je praštevilo")
```

Uporabili smo trik, ki ga večina drugih jezikov ne pozna: `else` za zanko `for`. Program pravi takole: za vsako število od `2` do `n-1` preveri, ali deli `n`. Če ga, povej, da `n` ni praštevilo in prekini zanko. Če se zanka konča po naravni poti (torej, če ne naleti na `break`), pa povej, da je število praštevilo.

Ne spreglejte, da je `else` poravnan s `for`, ne z `if`, torej se tudi nanaša na `for` in ne na `if`.

V večini drugih jezikov bi morali narediti nekaj v tem slogu.

```
n = int(input("Vnesi število: "))
ok = True
for i in range(2, n):
    if n % i == 0:
        ok = False
        break
if ok:
    print(n, "je praštevilo")
else:
    print(n, "ni praštevilo")
```

V začetku za število predpostavimo, da je praštevilo (`ok = True`), dokler mu ne dokažemo nasprotnega (*until proven guilty*). Čim mu najdemo delitelja, je krivda dokazana, `ok` postavimo na `False` in z `break` končamo zanko. Na koncu izpišemo, kar je potrebno.

Program bi deloval pravilno tudi brez `break`, vendar bo z njim hitrejši: ko enkrat ugotovimo, da število ni praštevilo, nima smisla preskušati naprej.

Dodatno ga pospešimo, če preskusimo le delitelje do korena iz `n`: `range(2, n)` zamenjamo z `range(2, math.floor(math.sqrt(n)) + 1)`. Če je `n` deljiv s kakim `d`, ki je večji od korena iz `n`, je deljiv tudi z k, ta pa je manjši od korena. Koren iz `n` zaokrožimo navzdol in prištejemo `1`.

V izogib dolgočasenju znalcev pokažimo še rešitev z izpeljanim seznamom. Število je praštevilo, če so vsi (`all`) ostanki po deljenju `n` z `i` različni od `0` (`n % i != 0`) za vse `i` od `2` do `n`.

```
def prastevilo(n):
    return all(n % i != 0 for i in range(2, n))
```

31. Vsota deliteljev

Precej podobna reč kot v prejšnjih dveh nalogah, le da deliteljev ne izpisujemo, temveč seštevamo.

```
def vsota_deliteljev(n):
    v = 0
    for i in range(1, n):
        if n % i == 0:
            v += i
    return v
```

Števec `i` smo spustili od `1` do `n - 1`, ker pač naloga tako hoče.

Nekoč bomo znali to napisati krajše:

```
def vsota_deliteljev(n):
    return sum(i for i in range(1, n) if n % i == 0)
```

32. Popolno število

Če smo dovolj pametni, da uporabimo, kar imamo, je funkcija trivialna: povedati mora, ali je število enako vsoti svojih deliteljev.

```
def popolno(n):
    return n == vsota_deliteljev(n)
```

Naloga nastavlja past tistim, ki ne razmišljajo o tem, kaj pomenijo pogoji. Daje jih namreč skušnjava, da bi napisali

```
def popolno(n):
    if n == vsota_deliteljev(n):
        return True
    else:
        return False
```

To deluje, vendar je čisto nesmiselno. Izraz `n == vsota_deliteljev(n)` že ima vrednost True ali False, torej natančno to, kar želimo vrniti.

Druga skušnjava je napačno razumeti namig. "Uporabiti, kar smo napisali v prejšnji nalogi" ne pomeni skopirati one funkcije

```
def popolno(n):
    v = 0
    for i in range(1, n):
        if n % i == 0:
            v += i
    return v == n
```

Tudi to deluje in tudi to je nesmiselno. Funkcije definiramo prav zato, da nam ne bi bilo potrebno večkrat pisati enih in istih stvari.

33. Vsa popolna števila

Potrebujemo le zanko, v kateri za vsako število vprašamo, ali je popolno.

```
for i in range(1, 1001):
    if popolno(i):
        print(i)
```

Čemu `range(1, 1001)` in ne `range(1, 1000)`? Kakor smo opozorili že parkrat (poslej pa ne bomo opozarjali več), `range` deluje tako, da je spodnja meja vključena, zgornja ne. Čemu je to dobra ideja, vam povesta učbenik in učitelj.

Če ne kličemo funkcije `popolno` (ta pa ne funkcije `delitelji`), se celoten program precej podaljša in zaplete.

34. Prijateljska števila

Ker smo si pripravili funkcijo vsota_deliteljev(n), je naloga preprosta.

```
def prijatelj(n):
    m = vsota_deliteljev(n)
    if vsota_deliteljev(m) == n:
        return m
```

Prijateljsko število je vsota deliteljev `n`. Shranimo jo v `m`. Preverimo, ali je vsota deliteljev `m`-ja enaka `n`. Če je, je `m` v resnici `n`-jev prijatelj in ga vrnemo.

Kje pa je `return None`? Ni ga. Funkcija, ki ne vrne ničesar, vrne `None`. Puritanci pravijo, da ni prav, da funkcije lahko včasih vračajo rezultat, včasih pa `None`. Če pa že to počnejo, pa mora biti jezik sestavljen tako, da nas prisili, da ob klicu takšne funkcije posebej poskrbimo za primer, ko vrne `None`. Najbrž imajo prav; med posebej atraktivnimi jeziki, ki upoštevajo ta nasvet, je Appleov jezik za iOS, Swift. A v Pythonu in podobnih jezikih pač razmišljamo drugače.

Če si ne bi upali klicati lastnih funkcij, temveč bi jih le prepisovali, bi to program precej zapletlo in podaljšalo.

```
def prijatelj(n):
    s = 0
    for i in range(1, n):
        if n % i == 0:
            s += i

    m = s
    s = 0
    for i in range(1, m):
    if m % i == 0:
        s += i

    if s == n:
        return m
```

35. Vsebuje 7

Naloga spominja na nalogo Števke. Pravzaprav je rešitev skoraj enaka, le namesto, da bi števke izpisovali, vrnemo `True` takoj, ko naletimo na kakšno sedmico. Če je ni, pa na koncu vrnemo `False`. Znani vzorec – zanka, znotraj nje pogoj, znotraj pogoja `return`.

```
def vsebuje_7(n):
    while n > 0:
        if n % 10 == 7:
            return True
        n //= 10
    return False
```

V nizom prijaznih jezikih – Python je eden njih – lahko pogoljufamo tako, da spremenimo število v niz in preverimo, ali tale niz vsebuje podniz 7. Pravzaprav lahko to naredimo v vseh jezikih, ki poznajo nize, vendar bi v nekaterih to izgledalo čudno, v Pythonu pa je povsem običajno:

```
def vsebuje_7(n):
    return "7" in str(n)
```

36. Poštevanka števila 7

Ker smo si pripravili funkcijo `vsebuje_7`, je naša naloga lahka.

```
def postevanka_7(n):
    for i in range(1, n + 1):
        if i % 7 == 0 or vsebuje_7(i):
            print("BUM")
        else:
            print(i)
```

Da bo šel `i` od 1 do `n`, uporabimo `range(1, n + 1)`. Za vsako število preverimo, ali je deljivo s 7 ali pa vsebuje sedmico (za kar imamo funkcijo `vsebuje_7`). Če je kaj od tega res, izpišemo BUM, sicer pa število.

Če ne bi imeli funkcije, bi bila stvar bolj zoprna.

```
def postevanka_7(n):
    for i in range(1, n + 1):
        bum = i % 7 == 0
        j = i
        while j and not bum:
            if j % 10 == 7:
                bum = True
            j //= 10
        if bum:
            print("BUM")
        else:
            print(i)
```

Imeli bomo spremenljivko bum. Povedala bo, ali je potrebno izpisati številko ali bum (glej zadnje štiri vrstice funkcije). Znotraj zanke, torej pri vsaki številki jo bomo najprej nastavili na `bum = i % 7 == 0`; torej bo `True`, če je število deljivo s 7. Nato preverimo, ali število vsebuje sedmico. To storimo tako, da zanko, kakršno smo imeli v funkciji `vsebuje_7`, prepišemo na mesto, kjer bi sicer klicali `vsebuje_7`. Namesto `return True` napišemo `bum = True` – odkrili smo števko 7, torej bomo izpisali BUM. Bodite pozorni tudi na pogoj v zanki: `while j and not bum`. Dodatek, `and not bum`, ni nujen, je pa smiseln: če že vemo, da bomo izpisali bum, se zanka lahko ustavi.

Čemu potrebujemo `j`? Če bi v zanki `while` mrcvarili `i`, dokler ga ne spravimo do ničle (ali ne odkrijemo sedmice), bi `print(i)` izpisal ničlo (ali kar ostane, ko pridemo do sedmice). Spremenljivko `i` moramo zato pustiti pri miru; njeno vrednost skopiramo v `j` in spreminjamo le-tega.

Na koncu izpišemo, kar je pač treba.

Program, v katerem smo uporabili funkcijo, je očitno precej enostavnejši. Problem smo razdelili na dva problema – določanje, ali števka vsebuje 7, smo izločili in ta del naloge rešili posebej. Tako v vsakem trenutku mislimo le na eno reč, kar je lažje. Pomožna funkcija ima tudi druge prednosti. V trenutku, ko odkrijemo, da je `i` deljiv s `7`, funkcija vrne `True` in konec. Z nobenimi postavljanji na `True` se nam ni treba ubadati. Pa še vrednosti `i`-ja ni potrebno prepisovati v `j`, saj ima funkcija svojo kopijo `i`-ja in ne bo spremenila "originala".

37. Fibonaccijevo zaporedje

Izkušnje s študenti prvega letnika kažejo, da je premetavanje spremenljivk, kakršnega zahteva naloga, nepričakovano trd oreh.

Začnimo z (za Python) običajno rešitvijo.

```
a = b = 1
for i in range(20):
    print(a)
    a, b = b, a + b
```

Da bi računali zaporedje, moramo imeti stalno pri roki zadnja dva člena. Shranili ju bomo v `a` in `b`; v `a` bo predzadnji člen, v `b` pa zadnji. Nato bomo v vsakem koraku izračunali "novi" predzadnji in zadnji člen. Predzadnji člen bo ta, ki je bil doslej zadnji (v `a` moramo prepisati, kar je bilo prej v `b`), zadnji člen pa bo vsota členov, ki sta bila prej predzadnji in zadnji. Z eno besedo (in v eni vrstici) `a`-ju in `b`-ju priredimo `b` in `a+b`.

Kdor ne razume, naj pred zadnjo vrstico doda

```
print("Prej: a: ", a, ", b: ", b)
```

za njo pa

```
print("Potem: a: ", a, ", b: ", b)
```

Tole se splača razumeti.

Vse skupaj je potrebno dvajsetkrat ponoviti, pred računanjem pa izpišemo vrednost `a`. (Zakaj ravno `a`? Zakaj pred računanjem in ne po njem? Poskusite izpisovati `b` ali pa zamenjajte zadnji vrstici, pa boste videli, da vam ne bo všeč.)

Zdaj pa pokažimo pogosto napačno rešitev.

```
# Ta program ne deluje!
a = b = 1
for i in range(20):
    print(a)
    b = a + b
    a = b
```

Logika programa je enaka kot prej, le napisana je narobe. V `b` izračunamo vsoto zadnjih dveh členov, `a`-ju pa priredimo, kar je bil prej `b`. Ha, kar je bil prej `b`? Žal `b` ni več, kar je bil prej, povozili smo ga z novo vrednostjo.

Tudi, če obrnemo prirejanji, ni nič boljše.

```
# Tudi ta program ne deluje!
a = b = 1
for i in range(20):
    print(a)
    a = b
    b = a + b
```

Ideja (nedelujoče) rešitve težave je v tem, da bi v `a` prepisali `b`, še preden `b` pokvarimo. Šele potem v `b` seštejemo, kar sta bila prej `a` in `b`. Ista pesem, le v drugem molu, kot bi rekel kapitan Vrungel, če bi znal programirati kaj boljše, kot je znal loviti veverice po norveških fjordih. Zdaj `a` ni več, kar je bil prej. Tule v `b` seštejemo `b` in `b`.

Če bi stvar programirali v, recimo, Javi ali Cju, bi naredili nekaj takšnega (v resnici bi morali dodati še kakšno vrstico in, predvsem, kakšen oklepaj, logika programa pa bi bila ista):

```
a = b = 1
for i in range(20):
    print(a)
    stari_a = a
    a = b
    b = stari_a + b
```

Spremenljivka `stari_a` je "začasna spremenljivka", v katero smo za trenutek shranili `a`, da nam je njegova vrednost na voljo vrstico kasneje. Takšne začasne spremenljivke so pogosta reč, zato jim radi rečemo `temp`, `tmp` ali `t`. Ime je pomembno; če je pametno izbrano, ga bo ta, ki bere program za nami (ali mi sami, ko ga bomo brali čez tri leta) lažje razumel.

Druga, zvita rešitev brez tretje spremenljivke je

```
a = b = 1
for i in range(20):
    print(a)
    b = a + b
    a = b - a
```

Kaj se bo v zadnji vrstici znašlo v `a`-ju? Ker je novi `b` vsota starega `b`-ja in trenutnega `a`-ja, bomo, če od novega `b`-ja odštejemo (trenutni) `a`, dobili ravno stari `b`. Grdo? Kar grdo, da. Lep program je temu, ki zna programirati, razumljiv brez tuhtanja. Pythonovski `a, b = b, a + b` je (vsaj za tiste, ki so vajeni Pythona) lepši.

38. Evklidov algoritem

Sledimo namigu:

```
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
```

Naloga je sicer iz klasičnega repertoarja, rešitev v Pythonu pa je še posebej lepa zaradi preprostosti, s katero dvema spremenljivkama priredimo dve vrednosti hkrati. Če programiramo v jeziku, kjer to ni mogoče, bomo naredili nekaj v slogu

```
def gcd(a, b):
    while b:
        ost = a % b
        a = b
        b = ost
    return a
```

Evklidov algoritem predpostavlja, da je `a` večji od `b`. Bi morali to preveriti in ju po potrebi pred zanko zamenjati? Kdor misli tako, naj si predstavlja, da je `a`, recimo 8 in `b` 13 ter premisli, kaj naredi prvi korak zanke.

Rešitev se splača primerjati z rešitvijo naloge, ki računa Fibonaccijeva števila. Zelo podobni sta. Tudi začetniki imajo z obema natančno isti problem. Premetavanje vrednosti med dvema spremenljivkama je iz neznanega razloga menda težko.

Zanke prek več reči hkrati

39. Indeks telesne teže

Tole je le preprosta vaja iz razpakiranja terk.

```
for ime, teza, visina in podatki:
    print(ime, teza / (visina / 100)**2)
```

40. Seštete trojke

Poleg tega, da moramo znati v zanki brati več reči naenkrat, se moramo spomniti starega vzorca – zanka, znotraj nje pogoj, znotraj pogoja `return`.

```
def trojke(s):
    for x, y, z in s:
        if x + y != z:
            return False
    return True
```

Kot vedno nalogah tega tipa: pazite, kam pišete zadnji `return`. Ne v zanko, temveč za njo.

V duhu funkcijskega programiranja lahko program zapišemo še veliko krajše. Kdor tega še ne zna, naj se ne vznemirja (razen, če je že čas, da bi znal).

```
def trojke(s):
    return all(x + y == z for x, y, z in s)
```

Pa če trojke niso urejene? V tem primeru imamo vsaj dve očitni rešitvi.

Prva: če niso urejene, jih pač uredimo. Če že znamo.

```
def neurejene_trojke(s):
    for t in s:
        x, y, z = sorted(t)
        if x + y != z:
            return False
    return True
```

Urejanje ni najcenejša stvar na svetu. (Ko programerji rečemo *cena* imamo v mislih porabo pomnilnika in procesorskega časa.) Kadar ni nujno, se mu izognemo, včasih pa je delo z urejenimi rečmi toliko hitrejše, da se splača žrtvovati nekaj časa za urejanje. Tule je očitna alternativa urejanju ta, da preverimo vse tri možne vsote.

```
def neurejene_trojke(s):
    for x, y, z in s:
        if x + y != z and x + z != y and y + z != x:
            return False
    return True
```

Pazite, tule uporabimo `and` in ne `or`! Razmisli! Če kaj pomaga pri meditaciji: kar smo napisali je (po de Morganovem pravilu) isto kot `if not (x + y == z or x + z == y or y + z == x)`.

Do daleč najboljše rešitve pa pridemo z uporabo zdrave pameti: če je vsota manjših dveh elementov enaka največjemu, je vsota vseh treh dvakrat tolikšna kot največji element (vsota vseh treh je vsota manjših dveh in večjega, kar je toliko kot dvakrat več od največjega). Jedro naše funkcije je torej

```
def neurejene_trojke(s):
    for trojka in s:
        if sum(trojka) != 2 * max(trojka):
            return False
    return True
```

Lepota te rešitve je v njeni splošnosti, saj ne deluje le za trojke, temveč tudi za seznane ali terke poljubne dolžine in pove, ali je v seznamu število, ki je vsota vseh ostalih.

41. Skalarni produkt

Naloga je pomembna za razlikovanje med gnezdenima zankama in zanko prek dveh seznamov. Veliko začetnikov bi namreč napisalo napačno rešitev.

```
# Napačna rešitev!
def skalarni (v, w):
    s = 0
    for e in v:
        for f in w:
            s += e * f
    return s
```

To množi vsak element `v` z vsakim elementom `w`. Ta, ki napiše takšen program, v resnici hoče reči nekaj drugega, namreč

```
# Napačna rešitev!
def skalarni(v, w):
    s = 0
    for e in v:
    for f in w:
        s += e * f
    return s
```

Prek obeh seznamov bi rad šel istočasno. Takole ne gre, tak program se niti ne začne izvajati, saj vsebuje sintaktično napako – koda znotraj prve zanke mora biti zamaknjena.

Vedno, kadar bi radi šli vzporedno prek dveh seznamov, uporabimo funkcijo `zip`, ki zadrgne dva seznama v seznam parov (ali več seznam v seznam terk). Vse terke pa, tako kot smo počei v prejšnjih nalogah, odpakiramo v par števil, `e` in `f`, ter dodajamo njun produkt v `s`, skalarni produkt.

```
def skalarni(v, w):
    s = 0
    for e, f in zip(v, w):
        s += e * f
    return s
```

Kdor zna, lahko nalogo reši z generatorskim izrazom.

```
def skalarni(v, w):
    return sum(e * f for e, f in zip(v, w))
```

Tisti, ki so vajeni kakega drugega jezika, bi morda naredili takole.

```
def skalarni(v, w):
    s = 0
    for i in range(len(v)):
        s += v[i] * w[i]
    return s
```

Množimo `i`-ti element v in `i`-ti element `w`, pri čemer `i` teče od 0, do koder je treba. Takšnih zank za zdaj še izogibamo, da se jih ne bi navadili uporabljati tudi takrat, ko so res res nepotrebne. Na vrsto pridejo v naslednjem poglavju, a še tedaj le, ko ne bo šlo drugače.

42. Ujemanja črk

Tako kot v prejšnji nalogi, tudi tule zadrgnemo oba niza v seznam parov črk. Za vsak par preverimo, ali je enak in v tem primeru povečamo števec enakih.

```
def st_ujemanj(b1, b2):
    uj = 0
    for c1, c2 in zip(b1, b2):
        if c1 == c2:
            uj += 1
    return uj
```

Funkcija `zip` tudi nima težav z različno dolgimi seznami ali besedami: tako kot bi šla prava zadrga, gre tudi `zip` toliko daleč, kolikor dolg je krajši od seznamov.

Profesionalec pa ob tej nalogi poreče samo

```
def st_ujemanj(b1, b2):
    return sum(c1 == c2 for c1, c2 in zip(b1, b2))
```

Primer običajne napačne rešitve, ki kaže na resno napako v dojemanju zanke `for`, je

```
def st_ujemanj(b1, b2):
    uj = 0
    for c1 in b1:
        for c2 in b2:
            if c1 == c2:
                uj += 1
    return uj
```

Tole primerja vsako črko `s1` z vsako črko `s2`. Kdor ne razume, kaj je narobe s tem programom, naj ne gre naprej, da ne bo eden mnogih, ki imajo probleme s tem! Požene naj tole

```
for c1 in "abcd":
    print("c1 je ", c1)
    for c2 in "XYWZ":
        print("    c2 je ", c2)
```

in meditira do razsvetljenja. Morda bo pomagal tudi tale premislek: verjetno smo namesto gornjega programa hoteli napisati tole

```
# Ta program ima sintaktično napako!
def st_ujemanj(b1, b2):
    uj = 0
    for c1 in b1:
    for c2 in b2:
        if c1 == c2:
            uj += 1
    return uj
```

Razlika je v tem, da zank nismo umaknili eno v drugo – hoteli bi z zanko po `b1` in istočasno po `b2`. To se ne da. Točneje, seveda se da, vendar ne na ta način. Kar hočemo doseči naredi, v poljubnem jeziku, funkcija na začetku, v jezikih, ki podpirajo funkcijsko programiranje (kaj je to, ni pomembno, pomembno je, da Python to podpira), pa tudi s funkcijo v slogu Pythonovega `zip`a (glej spodaj).

Enako napačna rešitev je

```
def st_ujemanj(b1, b2):
    uj = 0
    for i in range(len(b1)):
        for j in range(len(b2)):
            if b1[i] == b2[j]:
                uj += 1
    return uj
```

Natanko enako narobe deluje, le na bolj zapleten način je povedano.

Tisti, ki že znajo programirati v kakem drugem jeziku, bi morda napisali rešitev, ki ni napačna, vendar je v čudnem, nepythonovskem narečju.

```
def st_ujemanj(b1, b2):
    uj = 0
    for i in range(min(len(b1), len(b2))):
        if b1[i] == b2[i]:
            uj += 1
    return uj
```

Števec `i` mora od `0` do dolžine krajšega od nizov; če bi napisali le `for i in range(len(b1))`, program ne bi deloval, kadar bi bil niz `b1` daljši od niza `b2`.

43. Vzorec besede

Gremo po besedi in vzorcu. Če soležni (kako lep izraz so nam priskrbeli matematiki!) črki nista enaki in vzorec na tem mestu nima pike, se beseda ne ujema z vzorcem, zato vrnemo `False`. Če preživimo do konca vrnemo `True`, če sta niza enako dolga.

```
def se_ujema(beseda, vzorec):
    for b, v in zip(beseda, vzorec):
        if b != v and v != ".":
            return False
    return len(beseda) == len(vzorec)
```

Morda je kdo pričakoval, da bomo na koncu napisali

```
if len(beseda) == len(vzorec):
    return True
else:
    return False
```

To nima smisla. Izraz `len(beseda) == len(vzorec)` že ima vrednost `True` ali `False`. Sicer pa smo to že parkrat vzeli, ne?

Koga drugega morda moti, da se lotimo preverjanja posameznih znakov celo, kadar sta niza različno dolga. Čemu tega ne preverimo najprej? Lahko, lahko.

```
def se_ujema(beseda, vzorec):
    if len(beseda) != len(vzorec):
        return False
    for b, v in zip(beseda, vzorec):
        if b != v and v != ".":
            return False
    return True
```

Smo s tem pridobili kaj dosti časa? Bo funkcija zato kaj hitrejša? Morda, če so nizi res dolgi.

Profiji napišejo

```
def se_ujema(beseda, vzorec):
    return len(beseda) == len(vzorec) and \
        all(b == v or v == "." for b, v, in zip(beseda, vzorec))
```

44. Prva beseda

Gremo čez besede in za vsako preverimo, če se ujema z vzorcem. Funkcijo za to na srečo že imamo, ravno v prejšnji nalogi smo jo napisali. Če se, jo vrnemo.

Če ne bomo vrnili nobene besede, bomo vrnili `None`.

```
def prva_beseda(besede, vzorec):
    for beseda in besede:
        if se_ujema(beseda, vzorec):
            return beseda
```

Na konec funkcije bi lahko napisali `return None`, vendar ni potrebno. `None` vrnemo preprosto tako, da ne vrnemo ničesar.

Nekateri zmotno menijo, da je boljše tako:

```
def prva_beseda(besede, vzorec):
    for i in range(len(besede)):
        if se_ujema(besede[i], vzorec):
            return besede[i]
```

Ni boljše. Le daljše in nerodnejše.

Nekateri se bojijo spustiti zanko for čez prazen seznam, zato se prej prepričajo.

```
def prva_beseda(besede, vzorec):
    if not besede:
        return None
    for beseda in besede:
        if se_ujema(beseda, vzorec):
            return beseda
```

Po tem ni nobene potrebe. Če pa že morate preverjati (točneje: kadar morate preverjati – tule namreč res ni treba), ali je seznam prazen, pišite `if not besede` in ne `if len(besede) == 0` ali `if not len(besede)` ali `if besede != []`.

45. Paralelni skoki

Funkcija je nekoliko daljša, kot smo vajeni, a prav nič zapletena.

```
def paralelni_skoki(dolzine1, dolzine2):
    zmage1 = zmage2 = 0
    for skok1, skok2 in zip(dolzine1, dolzine2):
        if skok1 > skok2:
            zmage1 += 1
        elif skok2 > skok1:
            zmage2 += 1
        else:
            zmage1 += 0.5
            zmage2 += 0.5
    if zmage1 > zmage2:
        return 1
    elif zmage2 > zmage1:
        return 2
```

Gremo prek parov dolžin skokov. Če je daljši prvi, prištejemo zmago k prvi ekipi, če je daljši drugi skok, prištejemo zmago k drugi ekipi, sicer pa sta enaka in prištejemo pol točke vsaki ekipi.

Na koncu ugotovimo ali je večkrat zmagala prva ekipa in vrnemo 1, ali druga in vrnemo 2. Če ni zmagala nobena, ne vrnemo ničesar, torej vrnemo `None`.

Je res potrebno prištevati pol točke, kadar ne zmaga nobena od ekip? Ne, saj to na končni izid niti ne vpliva.

46. Mesto največjega elementa

Tule bomo uporabili funkcijo `enumerate`, da bomo dobili tako elemente seznama kot njuna mesta v seznamu.

```
def arg_max(s):
    najvecji = None
    naj_mesto = None
    for i, e in enumerate(s):
        if najvecji == None or e > najvecji:
            najvecji, naj_mesto = e, i
    return naj_mesto
```

Krajša rešitev je

```
def arg_max(s):
    return s.index(max(s))
```

V kakem drugem jeziku bi ob tem potožili, da mora računalnik dvakrat prek seznama: prvič išče največji element in drugič njegovo mesto. Vendar se tadva prehoda zanki izvajata znotraj funkcij `max` in `index`, ki sta sprogramirani v (veliko hitrejšem) jeziku C, medtem ko je zanka v prejšnjem programu napisana v Pythonu. Ta program zato ni le krajši temveč najbrž (odvisno od okoliščin) tudi hitrejši.

Tisti, ki se programiranja šele učijo, naj se metodi `index` raje izogibajo. Ne potrebujejo je velikokrat, predvsem pa se jim velikokrat zdi, da jo potrebujejo – in z njeno pomočjo rešijo nalogo na bolj zapleten način, kot bi bilo treba, ali pa celo narobe. Tule je rešitev z `index` sicer dobra, vendar ne bo pogosto tako.

Povejmo še, kako bi to rešili na nekoliko ne-Pythonovski način, z uporabo indeksov.

```
def arg_max(s):
    if not s:
        return None
    naj_mesto = 0
    for i in range(len(s)):
        if s[i] > s[naj_mesto]:
            naj_mesto = i
    return naj_mesto
```

S to rešitvijo ni nič narobe, vendar bomo, kadar bo šlo, raje uporabljali `enumerate`.

47. Olimpijske medalje

Upam, da smo si do te naloge zapomnili, kako imenitna reč je `enumerate`. Funkcija kot argument dobi drugi stolpec tabele (prejšnje rezultate), `enumerate` pa pripne prvega (letošnje). Da jih bomo začeli z `1`, ji bomo dodali še en argument, namreč `1`.

```
def napredek(s):
    gor = dol = 0
    for letos, prej in enumerate(s, 1):
        if letos < prej:
            dol += 1
        elif letos > prej:
            gor += 1
    return gor, dol
```

48. Vstavi teže

Tole je predvsem naloga iz spretnega dela z indeksi: eden se pomika naprej po seznamu imen, drugi pa se premakne na naslednjo težo le, ko se mora: vsakič, ko v prvem seznamu naletimo na žensko ime, vzamemo element iz drugega seznama.

```
def vstavi_teze(osebe, teze):
    i = 0
    for j, ime in enumerate(osebe):
        if ime[-1] != "a":
            osebe[j] = teze[i]
            i += 1
```

Navidez dobra ideja bi bila, da se znebimo drugega števca tako, da teže pobiramo iz seznama tež – recimo vzamemo in pobrišemo, z metodo `pop`.

```
def vstavi_teze(osebe, teze):
    for j, ime in enumerate(osebe):
        if ime[-1] != "a":
            osebe[j] = teze.pop(0)
```

Vendar le navidez. Seznama tež namreč ne smemo spreminjati. Pač pa lahko naredimo kopijo in spreminjamo le-to.

```
def vstavi_teze(osebe, teze):
    teze = teze.copy()
    for j, ime in enumerate(osebe):
        if ime[-1] != "a":
            osebe[j] = teze.pop(0)
```

49. Primerjanje seznamov

Klasična rešitev – v tem slogu bi, če spregledamo uporabo funkcije `zip`, pisali v jezikih, ki nimajo kakih posebno prijaznih funkcij za delo s seznami – je takšna.

```
def primerjaj(s, t):
    if len(s) != len(t):
        return 0
    p = 0
    for si, ti in zip(s, t):
        if si < ti:
            if p == 1:
                return 0
            p = -1
        elif si > ti:
            if p == -1:
                return 0
            p = 1
    return p
```

Spremenljivka `p` vsebuje možni izid. V začetku je 0, kar bo pomenilo, da sta seznama za zdaj enaka – naleteli nismo še na noben element, kjer bi se razlikovala. Če je `-1`, to pomeni, da so bili vsi elementi `s`-ja, ki so se razlikovali od istoležnih elementov `t`-ja, manjši. Če je, so bili vsi elementi `s`-ja večji od elementov `t`-ja.

Zdaj poglejmo, kaj storimo, ko naletimo na element `s`-ja, ki je manjši od elementa `t`-ja:

```
        if si < ti:
            if p == 1:
                return 0
            p = -1
```

Če je `p` enak `1`, smo že videli tudi elemente s-ja, ki so bili večji od elementov t-ja, torej moramo vrniti 0. Sicer je `p` enak 0 (različnih elementov še ni bilo) in postavimo ga na `-1` (`s` je manjši od `t`-ja) ali pa je enak `-1` in postavljanje na `-1` tako ali tako ne bo imelo vpliva.

Ravno obratno ravnamo, ko je element `s`-ja večji od `t`-ja.

Ko je zanke konec, vrnemo p. Ta bo -1 ali 1, če sta seznama različna ali 0, če nismo naleteli na nobeno razliko.

Pazite na `elif`: tule `else` ni čisto prava stvar, saj bi pokril tudi primer, ko sta `si` in `ti` enaka, tega pa nočemo.

Ko so študenti reševali tole nalogo na izpitu, so se lotili na prvi pogled preprosteje. Če sta seznama različno dolga ali pa popolnoma enaka, so vrnili 0. Sicer so prešteli, v koliko elementih je `s` manjši ali enak `t`-ju in v koliko elementih je večji ali enak. Na koncu so preverili, ali je manjši ali enak ali pa večji ali enak v vseh elementih.

```
def primerjaj(s, t):
    if len(s) != len(t) or s == t:
        return 0
    vec_s = vec_t = 0
    for si, ti in zip(s, t):
        if si >= ti:
            vec_s += 1
        if si <= ti:
            vec_t += 1
    if vec_s == len(s):
        return 1
    if vec_t == len(t):
        return -1
    return 0
```

Tole na koncu koncev niti ni preprostejše. In še eno past skriva: znotraj zanke ne smemo uporabiti `else` ali `elif`, saj se lahko zgodi, da sta resnična oba pogoja.

Malenkost elegantnejša je rešitev, ki prešteva le, ali je `s` kdaj manjši in ali je kdaj večji – brez "enak".

```
def primerjaj(s, t):
    if len(s) != len(t) or s == t:
        return 0
    vec_s = vec_t = 0
    for si, ti in zip(s, t):
        if si > ti:
            vec_s += 1
        elif si < ti:
            vec_t += 1
    if vec_s == 0:
        return -1
    if vec_t == 0:
        return 1
    return 0
```

Saj ne, da bi bilo kaj krajše, le duhovitejše je. Že ko sta za nami prvi dve vrstici, vemo, da seznama nista enaka. Če `s` ni bil nikoli večji, je bil vedno manjši ali enak. In obratno. Če je bil včasih večji in včasih manjši, pa vrnemo 0.

Rešitev se bistveno skrajša z malo spretne uporabe izpeljanih seznamov.

```
def primerjaj(s, t):
    if len(s) != len(t) or s == t:
        return 0
    if all(si < ti for si, ti in zip(s, t)):
        return -1
    if all(si > ti for si, ti in zip(s, t)):
        return 1
    return 0
```

S tistimi, ki rad razmišljajo in hočejo videti kak lep trik, premislimo tole jedrnato rešitev.

```
def primerjaj(s, t):
    v = [max(si, ti) for si, ti in zip(s, t)]
    return int(len(s) == len(t)) and (s == v) - (t == v)
```

V seznam `v` poberemo večje elemente iz `s` in `t`: sprehodimo se čez pare, `(si, ti)`, in iz vsakega para poberemo večji element. Če je s vedno vsebuje večje (ali enake) elemente, bo `s` enak `v`. Če `t` vsebuje večje (ali enake) elemente, bo `t` enak `v`.

Zdaj poglejmo `return` in za začetek prezrimo prvi del pogoja, vse do `and`. Spomnimo se še, da je `True` isto kot `1` in `False` isto kot `0`, torej lahko izraza `s == v` in `t == v` obravnavamo, kot da bi bila 0 ali 1. Če je `s` v resnici večji od `t` (tako, kot primerjanje definira ta naloga), bo `s == v` enak `1` in `t == v` bo enak `0` in funkcija bo res vrnila `1`. Če je večji `t`, je ravno obratno in funkcija vrne `-1`, kot mora. Če sta seznama enaka, so vsi trije seznami, `s`, `t` in `v` enaki in funkcija bo vrnila `1 - 1`, torej `0`, kar je spet ravno prav.

Če sta seznama različno dolga pa bi utegnili imeti problem: `zip` bo šel do dolžine krajšega od njiju. Če bi bil, recimo, `s` krajši, a večji (dokler traja, seveda), bi funkcija zmotno vrnila `1`. Zato potrebujemo še šaro pred `and`-om. Ta preveri, ali sta seznama enako dolga. Če nista, dobimo `False` in ga, da ustrežemo navodilom naloge, spremenimo v `0`, tako da pokličemo `int`. Če sta enako dolga, `True`, bomo po klicu `int` dobili `1`. Izraz `0 and <karkoli>` vrne `0`, izraz `1 and <karkoli>` pa `<karkoli>` - pa smo.

Seznami in nizi

50. Spol iz EMŠO

Malo znano dejstvo je, da se vsa trimestna števila, ki so večja ali enaka 500 (torej vsa števila od 500 do 999), začnejo s števko, ki je večja ali enaka 5. Kdor pozna ta fascinantni in šele nedavno dokazani izrek iz teorije števil (med študenti, ki so reševali izpit s to nalogo, jih žal ni bilo veliko), z nalogo nima veliko dela.

```
def je_zenska(emso):
    return emso[9] >= "5"
```

Ostali se lahko mučijo, recimo, takole

```
def je_zenska(emso):
    return int(emso[9:12]) >= 500
```

Precej manj želena je takšna rešitev.

```
def je_zenska(emso):
    if int(emso[9:12]) >= 500:
        return True
    else:
        return False
```

Take reči je žal pogosto videti. Izraz v pogoju, `int(emso[9:12]) >= 500`, se tako ali tako izračuna v `True` ali `False`, zato zapletanje z `if` in `else` ne služi ničemur. Pravzaprav se mora tisti, ki napiše kaj takega, vprašati, ali ne bi bilo bolj varno namesto

```
    if int(emso[9:12]) >= 500:
```

pisati

```
    if (int(emso[9:12]) >= 500) == True:
```

51. Pravilnost EMŠO

Tule gre le za spretnost. Začnimo z najgršim.

```
def preveri_emso(emso):
    vsota = 7 * int(emso[0]) + 6 * int(emso[1]) + 5 * int(emso[2]) \
          + 4 * int(emso[3]) + 3 * int(emso[4]) + 2 * int(emso[5]) \
          + 7 * int(emso[6]) + 6 * int(emso[7]) + 5 * int(emso[8]) \
          + 4 * int(emso[9]) + 3 * int(emso[10]) + 2 * int(emso[11])
    return (vsota + int(emso[-1])) % 11 == 0
```

Za zjokat. To je videti kot program najhujšega obupanca na izpitu. Takšnega, ki ni še nikoli slišal za zanke. Vsaj zanko naredimo.

```
def preveri_emso(emso):
    f = 7
    vsota = 0
    for e in emso[:-1]:
        vsota += f * int(e)
        f -= 1
        if f == 1:
            f = 7
    return (vsota + int(emso[-1])) % 11 == 0
```

Spremenljivke `f` se znebimo, če uporabimo `range(len(...))` in računamo faktor iz indeksa. Z malo premišljevanja vidimo, da `i`-to števko pomnožimo s `7 - i % 6`. To množi prvo s 7, drugo s 6 ... Ko pridemo do `i=6`, pa je `i % 6` enako 0 in ponovno množimo s 7.

```
def preveri_emso(emso):
    vsota = 0
    for i in range(len(emso[:-1])):
        vsota += (7 - i % 6) * int(emso[i])
    return (vsota + int(emso[-1])) % 11 == 0
```

Še lepše in bolj prav se naloge namesto z `range(len(...))` lotimo z `enumerate`.

```
def preveri_emso(emso):
    vsota = 0
    for i, e in enumerate(emso[:-1]):
        vsota += (7 - i % 6) * int(e)
    return (vsota + int(emso[-1])) % 11 == 0
```

52. Starost iz EMŠO

Najprej bomo izračunali, katerega dne, meseca in leta je oseba rojena. Za začetek se bomo delali, kot da je rojena leta 1xxx, torej bomo k trem števkam letnice prišteli 1000. Če bo vsota manjša od 1800, bomo dodali še 1000, ter tako poskrbeli za one, rojene po letu 2000. Starost bomo dobili tako, da bomo letnico rojstva odšteli od 2009. Za tiste, ki so rojeni januarja in to 26. ali prej, bomo k starosti dodali še 1.

```
def starost(emso):
    dan = int(emso[:2])
    mesec = int(emso[2:4])
    leto = 1000 + int(emso[4:7])
    if leto < 1800:
        leto += 1000
    starost = 2009 - leto
    if mesec == 1 and dan <= 26:
        starost += 1
    return starost
```

53. Domine

V rešitvi bomo uporabili klasičen trik z `zip`om. Če imamo seznam in bi radi primerjali zaporedne elemente tega seznama, za`zip`amo seznam s samim sabo brez prvega elementa, `zip(s, s[1:])`. Tako bo ničti element sparjen s prvim, prvi z drugim in tako naprej.

Gremo torej čez vse takšne pare in prvi element prve domine primerjamo z ničtim naslednje. Če se ne ujemata, zatulimo "Falš".

```
def domine(domine):
    for prej, potem in zip(domine, domine[1:]):
        if prej[1] != potem[0]:
            return False
    return True
```

Namesto trika z zipom lahko uporabimo tudi indekse. Ehm, dvojne indekse, saj imamo sezname seznamov oziroma sezname terk, parov.

```
def domine1(domine):
    for i in range(1, len(domine)):
        if domine[i - 1][1] != domine[i][0]:
            return False
    return True
```

Zanka teče od prve domine, ker vsako domino primerja s prejšnjo. Če bi namesto `range(1, len(domine))` pisali samo `range(len(domine))`, bi, po pravilih, ki veljajo za negativne indekse v Pythonu, primerjali še ničto (se pravi, prvo) domino z zadnjo. V tem primeru bi bilo to prav le, če bi bile domine zložene v krog. (Glej nalogo Ploščina poligona.)

V naprednejšem slogu nalogo rešimo takole

```
def domine(domine):
    return all(d1[1] == d2[2] for d1, d2 in zip(domine, domine[1:]))
```

Mogoče je nekoliko lepše tako:

```
def domine(domine):
    return all(d1 == d2 for (_, d1), (d2, _) in zip(domine, domine[1:]))
```

Funkcija `all` vrne `True`, če so resnični vsi elementi seznama, ki ga podamo kot argument, oziroma, v gornjem primeru, vsi elementi, ki jih daje podani generator.

54. Dan v letu

To (ogrevalno izpitno) nalogo so nekateri študenti zapletli do onemoglosti ... a le zato, ker se niso domislili, da bi shranili dolžine mesecev v seznam ali terko. Če pa se poleg tega domislimo še funkcije `sum`, pa je rešitev res kratka.

```
def dan_v_letu(dan, mesec):
    dni_v_mesecu = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
    return sum(dni_v_mesecu[:mesec - 1]) + dan
```

Sešteti moramo dneve, ki jih imajo meseci do trenutnega, k temu pa prišteti dan v trenutnem mesecu. Za 21. marec bomo sešteli `dni_v_mesecu[:2]`, torej 31+28 in še 21 zraven.

V resničnem življenju pogosto štejemo od 1; z meseci je že tako. Da bi bil prvi mesec, torej mesec 1, dolg 31 dni, lahko na začetek seznama dodamo še en, neobstoječi, mesec, ki ima 0 dni. Če storimo tako, v funkciji ni več potrebno odšteti 1 od meseca.

Poleg tega lahko prestavimo nastavljanje seznama pred funkcijo. Tako bo to storjeno enkrat za vselej in ne ob vsakem klicu funkcije.

```
dni_v_mesecu = (0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
def dan_v_letu(dan, mesec):
    return sum(dni_v_mesecu[:mesec]) + dan
```

55. Nepadajoči seznam

Gremo prek seznama – recimo z indeksom. Čim naletimo na element, ki je manjši od predhodnega, vrnemo `False`. Če takega ni, po zanki vrnemo `True`. Tako kot pri, recimo, praštevilih, le da tu še malo opletamo z indeksi.

```
def nepadajoc(s):
    for i in range(1, len(s)):
        if s[i] < s[i - 1]:
            return False
    return True
```

Popaziti moramo, da začnemo z elementom z indeksom 1, ne 0. Če bi pisali `for i in range(len(s))` bi primerjali tudi element 0 z elementom -1, torej zadnjim. (Razmislite, kako je videti nepadajoč seznam, v katerem je prvi element večji ali enak od zadnjega!)

Z zanko for moramo spet prek indeksov (`for i in range(1, len(s))`) namesto prek elementov (`for e in s`), ker potrebujemo indeks, da bomo izvedeli vrednost prejšnjega elementa, torej tistega, ki ima indeks `i - 1`. Pa je to res potrebno? Ni.

```
def nepadajoc(s):
    if not s:
        return True
    prejsnji = s[0]
    for trenutni in s:
        if trenutni < prejsnji:
            return False
        prejsnji = trenutni
    return True
```

Spremenljivki `prejsnji` in `trenutni` bosta vsebovali prejšnji in trenutni element seznama. Funkcija vrne `False`, če je trenutni manjši od prejšnjega. V začetku je prejšnji prvi (s[0]), stavek na koncu zanke, `prejsnji = trenutni`, pa je napisan s pogledom usmerjenim v prihodnost: kar je zdaj trenutni element, bo v naslednjem krogu prejšnji. Razumite to in razumeli boste frazo, ki jo boste srečali še velikokrat, tudi v tej zbirki.

Pozorni bralec je opazil, da smo zanko napisali nekoliko narobe. Bi ne bilo pravilno `for trenutni in s[1:]`? Ko je prejšnji element enak `s[0]`, ga moramo vendar primerjati z `s[1]`, ne `s[0]`! Bodimo praktični. Izraz `s[1:]` od Pythona zahteva, da pripravi nov seznam, enak prejšnjemu, le brez prvega elementa. Kot smo napisali, pa je povsem neškodljivo, le zanka naredi en krog več. V prvem krogu primerja prvi element s samim sabo; ker ni manjši od sebe, se pač ne zgodi nič. V drugem krogu pa že primerja ničti element s prvim, kar je prav.

Ta oblika je praktična, kadar v resnici ni indeksov. Z njo bi lahko iskali "nepadajoče datoteke":

```
def nepadajoca_datoteka(ime_dat):
    dat = open(ime_dat)
    prejsnja_vrsta = dat.readline()
    for trenutna_vrsta in dat:
        if trenutna_vrsta < prejsnja_vrsta:
            return False
        prejsnja_vrsta = trenutna_vrsta
    return True
```

Pravi Pythonovski način za reševanje takšnih nalog je zipanje seznama s samim sabo, tako kot smo storili že pri dominah. To nam da preprosto funkcijo.

```
def nepadajoc(s):
    for prejsnji, naslednji in zip(s, s[1:]):
        if naslednji < prejsnji:
            return False
    return True
```

Še več. Če znamo napisati tako, nam bo to pomagalo nekoč zapisati še krajše.

```
def nepadajoc(s):
    return all(naslednji >= prejsnji for prejsnji, naslednji in zip(s, s[1:]))
```

56. Mesta črke

Prva rešitev je podobna, kot bi bila v poljubnem programskem jeziku: z `i` od `0` do dolžine besede minus 1; če je `i`-ti znak enak iskanemu, dodamo `i` v seznam.

```
def mesta_crke(beseda, crka):
    mesta = []
    for i in range(len(beseda)):
        if beseda[i] == crka:
            mesta.append(i)
    return mesta
```

Za `range(len(...))` smo že povedali, da ga ne maramo, saj je lepše uporabiti `enumerate`. Kar navadite se ga.

```
def mesta_crke(beseda, crka):
    mesta = []
    for i, c in enumerate(beseda):
        if c == crka:
            mesta.append(i)
    return mesta
```

Če poznamo izpeljane sezname, pa isto reč povemo še veliko bolj jedrnato.

```
def mesta_crke(beseda, crka):
    return [i for i, c in enumerate(beseda) if c == crka]
```

57. Multiplikativni range

Poimenujmo argumente funkcije `e`, `fakt` in `dolz`. Storiti nam je tole: `e` bomo `dolz`-krat pomnožili s `fakt` in zmnožke zapisovali v seznam.

```
def mrange(e, fakt, dolz):
    xs = []
    for i in range(dolz):
        xs.append(e)
        e *= fakt
    return xs
```

Gre pa tudi hitreje, če malo razmislimo in če kaj znamo. Prvi element seznama je `e`, drugi `e*fakt`, tretji `e*fakt*fakt`, četrti `e*fakt*fakt*fakt` in ... `i`-ti element je `e*fakt**i` (dvojna zvezdica, vemo, pomeni potenciranje). Funkcijo bi lahko torej napisali tudi takole

```
def mrange(e, fakt, dolz):
    xs = []
    for i in range(dolz):
        xs.append(e * fakt**i)
    return xs
```

Kdor je vešč izpeljanih seznamov, to takoj preobrne v

```
def mrange(e, fakt, dolz):
    return [e * fakt**i for i in range(dolz)]
```

58. Sumljive besede

Rešitev zahteva le, da vemo, kaj je `split` in kaj `append`.

```
def sumljive(s):
    sumljivke = []
    for beseda in s.split():
        if 'u' in beseda and 'a' in beseda:
            sumljivke.append(beseda)
    return sumljivke
```

59. Kockarji

Potrebovali bomo tabelico, v katero bomo šteli, kolikokrat je kdo vrgel šestico. V začetku bo vsebovala `n` ničel, nato bomo šli prek niza in vsakič, ko vidimo šestico, povečali vsebino

ustreznega polja za 1. Katero pa je ustrezno polje? Preprosto, izračunamo ostanek po deljenju z `n`. Če imamo tri kockarje, se elementi 0, 3, 6, 9 nanašajo na ničtega, 1, 4, 7, 10 na prvega, 2, 5, 8, 11 na drugega.

Ko je tabelica sestavljena, poiščemo indeks elementa z največjo vrednostjo. Kot rezultat vrnemo številko povečano za 1, saj naloga hoče, naj ima prvi kockar zaporedno številko 1, ne 0.

```
def kockarji(s, n):
    sestic = [0] * n
    for i, m in enumerate(s):
        if m == 6:
            sestic[i % n] += 1
    naj_s = 0
    for i, s in enumerate(sestic):
        if  s > naj_s:
            naj_i, naj_s = i, s
    return naj_i + 1
```

60. Križanka

Sledimo namigu.

```
def se_ujemata(vzorec, beseda):
    if len(vzorec) != len(beseda):
        return False
    for i in range(len(vzorec)):
        if vzorec[i] != "." and beseda[i] != vzorec[i]:
            return False
    return True
```

Prvi pogoj je trivialen, besedi morata biti enako dolgi, sicer ne bo nič. Kdor se še opoteka, se bo morda opotekel ob pogoju v drugem `if`u. Tole pravi: če je črka iz vzorca pomembna (torej, če ni enaka piki) in ji črka iz besede ne ustreza (`beseda[i] != vzorec[i]`), vrnemo `False`.

Sicer pa funkcija spet ponavlja eno in isto kot vedno: v tej funkciji prepoznamo dva vzorca: prvi je zanka, ki gre vzporedno po dveh rečeh, kot v nalogi Ujemanja črk, in drugi funkcija, ki, tako kot v, recimo, Praštevila, vrne `False`, čim ji kaj ni prav, `True` pa le, če ji je prav vse, kar je videla v zanki. Začetni tečaj programiranja je le učenje uporabe (nepisanih, ohlapno definiranih) vzorcev.

Mimogrede, pogoj bi se dalo zapisati kompaktneje

```
        if beseda[i] != vzorec[i] != ".":
```

Ne počnite tega, nečitljivo je. Pogoji kot `a < b < c` ali `a == b < c == d`, so sprejemljivi, ker smo jih vajeni iz običajnega sveta, znamo jih prebrati (`a` in `b` sta enaka, `c` in `d` tudi, pri čemer sta prva dva manjša od drugih dveh). Čudnih kombinacij, kot gornja ali, še huje `a > b < c`, pa se izogibajmo.

Druga funkcija le kliče prvo in dodaja besede, ki se ujemajo.

```
def krizanka(vzorec, besede):
    ustrezne = []
    for beseda in besede:
        if se_ujemata(vzorec, beseda):
            ustrezne.append(beseda)
    return ustrezne
```

Vzorec, ki ga izvaja druga funkcija, je natanko tisto, kar počnejo izpeljani seznami. Funkcijo je zato preprosto napisati z njimi.

```
def krizanka(vzorec, besede):
    return [bes for bes in besede if se_ujemata(vzorec, bes)]
```

Zdaj pa rešimo nalogo še brez dodatne funkcije. Če hočemo to storiti kolikor toliko preprosto, moramo znati uporabiti `else` za `for`.

```
def krizanka(vzorec, besede):
    ustrezne = []
    for beseda in besede:
        if len(beseda) == len(vzorec):
            for i in range(len(vzorec)):
                if vzorec[i] != "." and beseda[i] != vzorec[i]:
                    break
            else:
                ustrezne.append(beseda)
    return ustrezne
```

Če naletimo na črko, v kateri se besedi ne ujemata (`beseda[i] != "." and beseda[i] != b[i]`), prekinemo zanko (`break`). Blok `else` se izvede le, če se zanka prekine; element bomo torej dodali le, če so se ujemale vse črke. Program je krajši kot prej, a bolj zapleten.

Če ne bi bilo `else`a za `for`, kot ga ni v večini jezikov, se reč zaplete še nekoliko bolj.

```
def krizanka(vzorec, besede):
    ustrezne = []
    for beseda in besede:
        if len(beseda) == len(vzorec):
            ok = True
            for i in range(len(beseda)):
                if vzorec[i] != "." and beseda[i] != vzorec[i]:
                    ok = False
                    break
            if ok:
                ustrezne.append(beseda)
    return ustrezne
```

Tale `ok` smo že videli – nikjer drugje kot v nalogi s Praštevili. Prav tam smo namreč programirali isti vzorec, le v drugi zgodbi.

Zelo narobe bi ravnal, kdor bi naivno poenostavil program tako:

```
# Tole je pošteno narobe
def krizanka(vzorec, besede):
    ustrezne = []
    for beseda in besede:
        if len(beseda) == len(vzorec):
            for i in range(len(beseda)):
                if vzorec[i] != "." and beseda[i] == vzorec[i]:
                    ustrezne.append(beseda)
    return ustrezne
```

Ta program doda besedo med ustrezne, čim se ujema z vzorcem *v vsaj eni* pomembni črki. In, še huje, doda jo tolikokrat, kot se ujema. (Če kdo misli, da bi se problem rešilo z `break`, za `append`om, misli napačno. S tem bi dosegli le, da se beseda doda le enkrat, še vedno pa bi se dodala, če bi se ujemala v vsaj eni črki).

Kot se ne spodobi, so tu še rešitve v eni vrstici. Prva uporablja funkcijo `all`: kaj dela, lahko uganete ali pa si ogledate v dokumentaciji.

```
def krizanka(vzorec, besede):
    return [beseda for beseda in besede if len(beseda) == len(vzorec) and all(c1
== '.' or c1 == c2 for c1, c2 in zip(vzorec, beseda))]
```

Druga, nekoliko neučinkovitejša in grša rešitev (a zabavna v svoji grdoti) se ji izogne.

```
def krizanka(vzorec, besede):
    return [beseda for beseda in besede if len(beseda) == len(vzorec) == sum(c1
in c2+"." for c1, c2 in zip(vzorec, beseda))]
```

Pogoj pravi, da mora biti dolžina vzorca enaka dolžini besede in številu ujemajočih se črk, pri čemer se črki ujemata, če je črka iz vzorca enaka črki iz besede ali piki.

61. Sosedi

Naredimo nov, prazen seznam, v katerega bomo naračunali rezultat. Gremo od začetka do (skoraj) konca seznama in v vsakem koraku v novi seznam dodamo vsoto prejšnjega in naslednjega elementa.

```
def stevilo_sosedov(s):
    res = []
    for i in range(len(s) - 1):
        res.append(s[i - 1] + s[i + 1])
    res.append(s[0] + s[-2])
    return res
```

Popaziti je potrebno le na robove. Ko je `i` enak 0, s tem ni težav, saj z `s[-1]` dobimo ravno zadnji element. Ko bi bil `i` na koncu seznama, pa bi z `s[i + 1]` dobili napako. In tega nočemo. ;) Zato smo pustili `i` le do `len(s) - 1` in zadnji element dodali šele na koncu.

Druga rešitev je uporaba modula, s katerim se `i + 1`, ko je `i` že enak dolžini seznama, spremeni v `i + 1` spremeni v `0`.

```
def stevilo_sosedov(xs):
    ys = []
    for i in range(len(xs)):
        ys.append(xs[i - 1] + xs[(i + 1) % len(xs)])
    return ys
```

Kdor ve za izpeljane seznama, lahko nalogo reši takole:

```
def stevilo_sosedov(s):
    return [s[i - 1] + s[i + 1] for i in range(len(s) - 1)] + [s[0] + s[-2]]
```

ali takole

```
def stevilo_sosedov(s):
    return [s[i - 1] + s[(i + 1) % len(s)] for i in range(len(s) - 1)]</xmp>
```

62. Glajenje

Dolgočasna rešitev je zanka, ki gre od prvega elementa do predpredpredzadnjega; vsakič izračuna vsoto štirih elementov, jo deli s 4 in doda v nov seznam.

```
def glajenje(s):
    novi = []
    for i in range(len(s) - 3):
        novi.append(sum(s[i:i+4]) / 4)
    return novi
```

Ko bomo znali, bomo isto reč napisali krajše:

```
def glajenje(s):
    return [sum(s[i:i+4]) / 4 for i in range(len(s) - 3)]
```

Premislimo, koliko števil bo morala sešteti tako napisana funkcija. Če pozabimo na začetek in konec: vsako število se pojavi v štirih izračunanih poprečjih. Če ima seznam 100 števil, bo funkcija seštela 400 števil – vsakega štirikrat.

In na tem mestu storimo, kar tudi sicer nikoli ne škodi: vključimo možgane. Da bo lažje, namesto poprečij za začetek opazujmo vsote. Vsota prvih štirih je enaka `v0 = s[0] + s[1] + s[2] + s[3]` (ime `v0` smo ji dali, da se bomo lažje pogovarjali). Vsota drugih štirih je `v1 = s[1] + s[2] + s[3] + s[4]`, to pa je slučajno ravno `v1 = v0 – s[0] + s[4]`. Jasno, vsota elementov od prvega do četrtega je taka kot vsota od ničtega do tretjega minus ničti plus četrti. In tako naprej, `v2 = v1 – s[1] + s[5]`. Ko računamo poprečja, moramo le še vse skupaj deliti s 4. Tako pridemo do naslednje funkcije.

```
def glajenje(s):
    if not s:
        return []
    novi = [sum(s[:4]) / 4]
    for i in range(len(s) - 4):
        novi.append(novi[-1] + (-s[i] + s[i + 4]) / 4)
    return novi
```

V seznam tekočih popręčij za začetek damo poprečje prvih štirih elementov. Nato v zanki (ki jo ponovimo enkrat manj kot zgoraj: da mora biti tako, vidimo po indeksu `i + 4`, sklepamo pa lahko tudi po tem, da mora zanka dodati en element manj, kot zgoraj, saj seznam `n` dobi prvi element že pred zanko) vsakič odštejemo `i`-ti element in prištejemo `i+4`-tega, ki ju, ker računamo poprečja in ne vsot, delimo s `4`.

63. An ban pet podgan

Najprej slabša rešitev, za manj razmišljujoče. V začetku kažemo na osebo z indeksom 0. Cela izštevanka ima enajst zlogov, torej se od trenutne osebe desetkrat pomaknemo na naslednjo (le desetkrat zato, ker ob prvem zlogu pokažemo na trenutno osebo). Če se pomaknemo prek zadnje, se vrnemo na prvo. Oseba, na katero kažemo po desetih premikih, gre ven. Osebe za njo se pomaknejo za eno mesto nižje (ko izštevamo Majo, Janjo in Sabino, v prvem krogu izločimo Janjo; Sabina gre na njeno mesto, če bi bil za Sabino še kdo, bi se pomaknil na njeno mesto...) Če smo izločili zadnjo osebo, se moramo pomakniti na prvo. Vse skupaj ponavljamo, dokler ne ostane le še ena oseba.

```
def anban(osebe):
    osebe = osebe[:]
    trenutna = 0
    while len(osebe) > 1:
        for i in range(10):
            trenutna += 1
            if trenutna == len(osebe):
                trenutna = 0
        del osebe[trenutna]
        if trenutna == len(osebe):
            trenutna = 0
    return osebe[0]
```

Čemu služi `osebe = osebe[:]` na začetku? S tem naredimo kopijo seznama. Iz seznama bo funkcija brisala elemente in prav je, da to počne s svojo kopijo, ne z originalom.

Funkcija deluje, ni pa posebej učinkovita. Da bomo lažje prišli do "prave" rešitve, program majčkeno spremenimo.

```
def anban(osebe):
    osebe = osebe[:]
    trenutna = 0
    while len(osebe) > 1:
        for i in range(10):
            trenutna = (trenutna + 1) % len(osebe)
        del osebe[trenutna]
    return osebe[0]
```

Razmislite, pa boste videli, da je to pravzaprav isto. Tisti `% len(osebe)` le postavi `trenutna` na `0`, če je že dosegel število oseb. Tako smo se znebili onih zoprnih pogojev, dobili pa tudi namig za naprej.

Namesto, da bi k `trenutna` desetkrat prišteli ena, le enkrat prištejmo deset (matematiki nas namreč prepričujejo, da je to eno in isto).

```
def anban(osebe):
    osebe = osebe[:]
    trenutna = 0
    while len(osebe) > 1:
        trenutna = (trenutna + 10) % len(osebe)
        del osebe[trenutna]
    return osebe[0]
```

Tudi tistega premikanja trenutne osebe za brisanjem smo se znebili, saj bo `% len(osebe)` poskrbel tudi za to, da `trenutna` ne bo kazal onstran gornje meje seznama.

64. Največji skupni delitelj seznama

Naloga je, zanimivo, presenetila precej študentov na izpitu, dasiravno je na moč preprosta, saj ne zahteva nič drugega kot tisto, kar bi naredili tudi ročno, če bi morali poiskati največji skupni delitelj več števil. Pa tudi navodila so jasna: poiščeš največji skupni delitelj prvih dveh, potem največji skupni delitelj teh dveh in tretjega (tako dobiš največji skupni delitelj prvih treh), potem največji skupni delitelj prvih treh in četrtega...

V spremenljivki `delitelj` bomo hranili največji skupni delitelj vseh števil, ki smo jih pregledali doslej. Za začetek ji bomo priredili kar prvo število, češ, pregledali smo eno samo število in njegov največji delitelj je kar število samo. Funkcijo, ki poišče največji skupni delitelj, pa smo napisali že v nalogi Evklidov algoritem.

```
def evklid(a, b):
    while b:
        a, b = b, a % b
    return a

def skupni_delitelj(s):
    delitelj = s[0]
    for e in s[1:]:
        delitelj = evklid(e, delitelj)
    return delitelj
```

Funkcijo smo napisali tako, da začne pregledovanje seznama števil (zanka `for`) pri drugem številu (`s[1:]`). To je pravzaprav brez potrebe: tudi če bi pregledali ves seznam, bi dobili isto, saj bi v prvem krogu `ekvlid` ugotovil, da je največji skupni delitelj prvega števila in prvega števila pač prvo število.

Če bo bralec kdaj vedel za Pythonove funkcije, s katerimi je mogoče programirati v funkcijskem slogu, pa bo lahko nalogo rešil kar takole:

```
from functools import reduce
def skupni_delitelj(s):
    return reduce(evklid, s)
```

65. Oškodovani otroci

Najpreprostejša rešitev gre prek vse otrok, od ničtega do petega, in za vsakega preveri, ali je med dobitniki bombonov. Če ni, funkcija vrne `False`, sicer preveri naslednjega. Če ne naleti na nikogar, ki ni ostal brez bombonov, vrne `True`.

```
def imajo_vsi(delitev):
    for otrok in range(6):
        if otrok not in delitev:
            return False
    return True
```

Bolj izobraženi bi se zmrdovali, da mora Python vsakič, ko ga vprašamo "`otrok not in delitev`", preleteti ves seznam – ali vsaj toliko, da najde iskanega otroka. Če se hočemo temu izogniti, storimo raje takole: pripravimo si seznam, ki bo imel toliko `False`-ov, kolikor je otrok, in to bo pomenilo, da še noben otrok ni dobil bombona. Nato gremo čez seznam dobitnikov in ko, recimo, vidimo, da je drugi otrok dobil bombon, spremenimo drugi element onega seznama v `True`. Na koncu preverimo, ali je v seznamu kakšen `False`; če ga najdemo, vrnemo `False`, sicer `True`.

```
def imajo_vsi(delitev):
    ima_bombon = [False] * 6
    for otrok in delitev:
        ima_bombon[otrok] = True

    for e in delitev:
        if not e:
            return False
    return True
```

Python ima funkcijo `all`, ki smo jo v prejšnjih dveh nalogah videli v nekoliko bolj zapleteni vlogi, tule pa ju lahko opišemo nekoliko poenostavljeno: funkcija dobi seznam reči in vrne 2, če so vse reči resnične. Otroke lahko torej spišemo tudi tako:

```
def imajo_vsi(delitev):
    ima_bombon = [False] * 6
    for otrok in delitev:
        ima_bombon[otrok] = True
    return all(ima_bombon)
```

Namesto, da beležimo le, ali otrok ima bombon ali ne, lahko preštejemo, koliko bombonov ima kdo in vrnemo `False`, če jih ima kdo `0`. Ker je 0 neresnična, lahko tudi v tej obliki uporabimo `all`.

```
def imajo_vsi(delitev):
    ima_bombon = [0] * 6
    for otrok in delitev:
        ima_bombon[otrok] += 1
    return all(ima_bombon)
```

Ta rešitev je koristna, ker vodi v še krajšo: Python ima modul collections, v katerem je funkcija Counter, ki je namenjena štetju reči. Counter bo naredil točno to, kar delajo prve tri vrstice naše funkcije. Vse skupaj lahko torej skrčimo v

```
from collections import Counter

def imajo_vsi(delitev):
    return all(Counter(delitev))
```

Če že kažemo naprednejše rešitve, pokažimo še eno. Seznam pretvorimo v množico in pogledamo koliko elementov ima. Če jih ima 6, to pomeni, da je bombone dobilo šest različnih otrok – torej vsi.

```
def imajo_vsi(delitev):
    return len(set(delitev)) == 6
```

66. Lomljenje čokolade

Tule je bilo potrebno najprej malo premisliti: ali lomimo čokolado z leve ali z desne, je vseeno. Potrebno je le opazovati, koliko vrstic in koliko stolpcev ostane.

```
def cokolada(n, odlomki):
    sirina = visina = n
    for odlomek in odlomki:
        kje, koliko = odlomek[0], int(odlomek[:1])
        if kje in "<>":
            sirina -= koliko
        else:
            visina -= koliko
    return max(0, sirina) * max(0, visina)
```

Bodimo pozorni na to, kako elegantno razpakirati niz: vzeti moramo prvo črko (`odlomek[0]`) in ostanek (`odlomek[:1]`). Ostanek pretvorimo v število in da bi bilo bolj očitno, da gre za "razpakiranje" niza, obe prirejanji opravimo kar naenkrat - `kje` in `koliko` sta prvi znak niza in ostanek niza.

Nato zmanjšamo bodisi širino bodisi višino za koliko. Nekateri ste pred zmanjševanje dodali `if` in v primeru, da odtrgamo več, kot imamo, le nastavili širino oz. višino na 0. Tule smo ga lomili naprej in šele na koncu poskrbeli za primer, ko čokolade zmanjka.

Na koncu se moramo upreti skušnjavi, da bi pisali

```
if sirina * visina < 0:
    return 0
```

To ne deluje, kadar sta oba, širina in višina, manjša od 0. Če od čokolade velikosti 10x10 odlomimo 11 vrstic in 11 stolpcev, bomo imeli `(-1) * (-1) = 1` košček? Prelepo, da bi bilo res. :)

Namesto tega bi morali pisati

```
if sirina < 0 or visina < 0:
    return 0
else:
    return sirina * visina
```

ali kako različico tega. V gornji rešitvi smo uporabili `max(0, sirina)`. Če je `sirina` večja od 0, bo `max(0, sirina)` kar `sirina`; če je `sirina` negativna, pa bo `max(0, sirina)` vrnil `0`.

Malo zapakirana rešitev je

```
def cokolada(n, odlomki):
    return max(0, n - sum(int(x[1:]) for x in odlomki if x[0] in "<>")) \
        * max(0, n - sum(int(x[1:]) for x in odlomki if x[0] in "v^"))
```

67. Razcep na praštevila

Funkcijo za določanje praštevilskosti smo že prežvečili, tule jo le ponovimo.

```
def prastevilo(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Vsa praštevila med od `2` do `n` izvemo tako, da v zanki kličemo gornjo funkcijo in vsa praštevila pridno zlagamo v seznam.

```
def prastevila(n):
    res = []
    for i in range(2, n+1):
        if prastevilo(i):
            res.append(i)
    return res
```

Hitrejša bi bila rešitev z izpeljanim seznamom.

```
def prastevila(n):
    return [i for i in range(2, n+1) if prastevilo(i)]
```

Deljivost zapišemo skladno z namigom.

```
def deljivost(n, i):
    k = 0
    while n % i == 0:
        k += 1
        n //= i
    return k
```

Duhovito bližnjico uberemo, če opazimo, da se `k` povečuje kot kak števec v zanki `for`, poleg tega pa vemo, da nobeno celo število ne more deliti `n`-ja več kot `n`-krat.

```
def deljivost(n, i):
    for k in range(n):
        if n % i > 0:
            return k
        n //= i
```

Za konec zapišimo še funkcijo za razcep, ki uporablja gornje funkcije.

```
def razcep(n):
    delitelji = []
    for p in prastevila(n):
        d = deljivost(n, p)
        if d > 0:
            delitelji.append((p, d))
    return delitelji
```

Slednjo funkcijo bi resen programer najbrž zapisal v eni vrstici.

```
def razcep(n):
    return [(p, deljivost(n, p)) for p in prastevila(n) if n % p == 0]
```

68. Pari števil

Najprej poglejmo, kako dobimo vsoto števk števila. En način je, da si pripravimo pomožno funkcijo, ki kot argument dobi število in kot rezultat vrne vsoto števk.

```
def vsota_stevk(i):
    v = 0
    for c in str(i):
        v += int(c)
    return v
```

Številko smo pretvorili v niz. Ko gremo z zanko čez niz, dobivamo posamezne števke, te pretvarjamo nazaj v števila in jih seštevamo. (Podobno reč, le malo bolj zapleteno, smo počeli v nalogi Obrnjena števila.)

Gre pa tudi hitreje. Vsoto števk števila `i` lahko dobimo preprosto s

```
sum(int(c) for c in str(i))
```

ali

```
sum(map(int, str(i))
```

Preostanek je preprost: dve zanki, ena od 1 do 100, druga pa od števca prve zanke pa do 1000. Da bi spustili obe zanki do 1000, ni nobene potrebe, saj morajo imeti števila različno število mest, torej mora biti vsaj eno število 100 ali manj. (Čemu pa do 100, ne do 99? Da ne izpustimo para 100, 1000!). Za vsak par preverimo, ali ima različno število števk (obe števili pretvorimo v niza in primerjamo njuno dolžino) in enako vsoto. Če to drži, ju dodamo v seznam.

```
def pari():
    p = []
    for i in range(1, 101):
        for j in range(i + 1, 1001):
            if len(str(i)) != len(str(j)) and vsota_stevk(i) == vsota_stevk(j):
                p.append((i, j))
    return p
```

Da bi dodali isti par dvakrat se nam pač ne more zgoditi: `i` je vedno manjši od `j`, v vseh parih, ki jih dobimo, bo prvi element manjši.

V gornjem primeru smo uporabili pomožno funkcijo vsota_stevk, enako dobro bi bilo tudi brez nje.

```
def pari():
    p = []
    for i in range(1, 1001):
        for j in range(i + 1, 1001):
            if len(str(i)) != len(str(j)) and sum(map(int, str(i))) ==
sum(map(int, str(j))):
                p.append((i, j))
    return p
```

Veliko hitrejše pa je, če uporabimo nekaj trikov. Pripravimo si lahko tabelo, v kateri za vsako številko vnaprej izračunamo vsoto njenih števk. Na kakšen način v spodnji rešitvi zoptimiziramo zanke, pa odkrijte sami.

```
def pari():
    ss = [sum(map(int, str(i))) for i in range(1001)]
    s = [(100, 1000)]
    for i in range(10, 100):
        if ss[i] < 10:
            s.append((ss[i], i))

    for i in range(100, 1001):
        if ss[i] < 10:
            s.append((ss[i], i))
        for j in range(10, 100):
            if ss[i] == ss[j]:
                s.append((j, i))
    return s
```

Takšna rešitev je veliko veliko hitrejša od prve, vendar pri hitrosti pridobimo predvsem po zaslugi shranjenih vsot števk, ne trikov z zankami `for`.

69. Ploščina poligona

Naloga bi bila preprostejša, ko ne bi bi bilo treba misliti na zadnje oglišče, ki ga je potrebno povezati s prvim – zadnji člen v gornji vsoti. Lahko ga obravnavamo posebej, tako da na koncu prištejemo zmnožek, ki se nanaša na koordinate vozlišč `0` in `n-1`. V resnici pa v Pythonu to niti ni potrebno: spodnja funkcija deluje povsem pravilno, le da zadnji člen postavi na začetek: najprej vzame člena z indeksoma `0` in `-1`, kar je isto kot `0` in `n-1`.

```
def ploscina(ogl):
    p = 0
    for i in range(len(ogl)):
        p += (ogl[i][0]+ogl[i-1][0])*(ogl[i][1]-ogl[i-1][1])
    return abs(p)/2
```

Zapisati taisti račun v eni vrstici ni nič zahtevnega.

```
def ploscina(ogl):
    return abs(sum((ogl[i][0]+ogl[i-1][0])*(ogl[i][1]-ogl[i-1][1]) for i in
range(len(ogl))))/2
```

70. Sodost čebel

Pripravimo dva prazna seznama, `sode` in `lihe`. Nato gremo čez podani seznam in vržemo ime vsake čebele v seznam, v katerega sodi.

```
def loci(cebele):
    sode = []
    lihe = []
    for ime, koliko in cebele:
        if koliko % 2 == 0:
            sode.append(ime)
        else:
            lihe.append(ime)
    return sode, lihe
```

Paziti je potrebno le, da ne pišemo `lihe = sode = []`, saj bosta `lihe` in `sode` potem en in isti seznam z dvema imenoma: kar vstavimo v enega, vstavimo v oba, saj je to isti seznam.

Zdaj pa še malo naprednejše telovadbe.

Vsebino zanke lahko skrajšamo tako, da z `if-else` določamo, v kateri seznam bomo dodajali.

```
def loci(cebele):
    sode = []
    lihe = []
    for ime, koliko in cebele:
        (sode if koliko % 2 == 0 else lihe).append(ime)
    return sode, lihe
```

V počasnem posnetku: izraz `sode if koliko % 2 == 0 else lihe` ima vrednost `sode`, če je `koliko` sod, in `lihe`, če je lih. Poklicati moramo `append` za ta seznam (za to ga moramo zapreti v oklepaj – če bi pisali `sode if koliko % 2 == 0 else lihe.append(ime)`, bi se `append`

nanašal na `lihe`, saj bi Python to prebral `lihe.append`). Torej bo `append` dodajal v ta ali v oni seznam.

Pravzaprav pa tudi `if-elsa` ne potrebujemo:

```
def loci(cebele):
    sode = []
    lihe = []
    for ime, koliko in cebele:
        [sode, lihe][koliko % 2].append(ime)
    return sode, lihe
```

Seznama `sode` in `lihe` zložimo v nov seznam, `[sode, lihe]`. Rezultat izraza `koliko % 2` je `0` ali `1`, torej bo `[sode, lihe][koliko % 2]` pobral `0`-ti ali `1`-vi element seznama `[sode, lihe]`. Če je spet potreben počasni posnetek: recimo, da je `koliko` enak `14`. V tem primeru je `koliko % 2` enako `0` in gornji izraz se prevede v `[sode, lihe][0]`. Ničti element seznama `[sode, lihe]` je seznam `sode`. In v tega dodamo ime čebele.

Navsezadnje pa tudi ne potrebujemo dveh imen za seznama.

```
def loci(cebele):
    sode_lihe = ([], [])
    for ime, koliko in cebele:
        sode_lihe[koliko % 2].append(ime)
    return sode_lihe
```

Pokažimo še rešitev z izpeljanimi seznami:

```
def loci(cebele):
    sode = [ime for ime, koliko in cebele if koliko % 2 == 0]
    lihe = [ime for ime, koliko in cebele if koliko % 2 == 1]
    return sode, lihe</xmp>
```

71. Sodi – lihi

Najprej razmislimo, kako preveriti, ali je par zaporednih elementov v seznamu, označimo ju z `s[i]` in `s[i+1]`, "pravilen", torej takšen, da je eno od števil sodo, drugo liho. Napisali bi lahko

```
s[i] % 2 == 0 and s[i + 1] % 2 == 1  or  \
s[i] % 2 == 1 and s[i + 1] % 2 == 0
```

se pravi, ali je prvo sodo in drugo liho ali pa prvo liho in drugo sodo. Ta, ki se mu zdi to lepo, bo moral zamenjati okulista. Gre namreč tudi veliko preprosteje: števili preprosto seštejemo. Če je rezultat sod, sta bodisi obe lihi bodisi obe sodi, česar ne maramo. Če je lih, pa je eno liho in drugo sodo.

Zdaj, ko to vemo, je preprosto in zelo podobno nalogi Nepadajoči seznam.

```
def sodi_lihi(s):
    for i in range(len(s)-1):
        if (s[i] + s[i + 1]) % 2 == 0:
            return False
    return True
```

Rešitev s trikom: če je seznam prazen ali pa ima le en element, je pravilen. Sicer pa delamo tole: računamo, kakšen je ostanek vsot po dveh zaporednih členov pri deljenju z 2. Lahko je 0 ali 1. Hočemo, da so vsi 1, se pravi, da mora biti minimum ostankov enak 1.

```
def sodi_lihi_a(s):
    return len(s) < 2 or min((i + j) % 2 for i, j in zip(s, s[1:])) == 1
```

72. Najprej lihi

Najprej dolgočasna rešitev. Pripravimo dva seznama, enega za lihe, drugega za sode. Gremo čez podani seznam in mečemo liha števila v enega, soda v drugega. Nato zamenjamo vsebino podanega seznama z vsoto prvega in drugega.

```
def najprej_lihi(s):
    lihi = []
    sodi = []
    for e in s:
        if e % 2 == 0:
            sodi.append(e)
        else:
            lihi.append(e)
    s[:] = lihi + sodi
```

Da se ne bo kdo pritoževal, da je naloga dolgočasna in, predvsem, neduhovita, se znebimo `if`-a. Oba seznama mimogrede postavimo v nov seznam in append-ajmo bodisi v enega bodisi v drugega.

```
def najprej_lihi(s):
    lihi = []
    sodi = []
    for e in s:
        [sodi, lihi][e % 2].append(e)
    s[:] = lihi + sodi
```

Zdaj je `[sodi, lihi][e % 2]` isto kot `sodi`, če je `e % 2` enako `0`, in isto kot `lihi`, če je `e % 2` enako 1.

Namesto da bi `sodi` in `lihi` vsakič znova tlačili v seznam (ki smo ga naredili le zato, da smo se znebili `if`-a), lahko kar ves čas delamo s skupnim seznamom.

```
def najprej_lihi(s):
    sodilihi = [[], []]
    for e in s:
        sodilihi[e % 2].append(e)
    s[:] = sodilihi[1] + sodilihi[0]
```

Zdaj pa lepše rešitve. Če poznamo izpeljane sezname, brez težav sestavimo seznam lihih in seznam sodih števil ter ju seštejemo. V enem zamahu.

```
def najprej_lihi(s):
    s[:] = [x for x in s if x % 2 == 1] + [x for x in s if x % 2 == 0]
```

Zmagovalna rešitev pa je ta: seznam s preprosto uredimo po ostanku po deljenju z 2.

```
def najprej_lihi(s):
    s.sort(key=lambda x: x % 2, reverse=True)
```

Ali, enako lepo,

```
def najprej_lihi(s):
    s.sort(key=lambda x: 1 - x % 2)
```

73. Sodo in liho in sodo in liho

Pripravimo seznam lihih in seznam sodih števil. Nato gremo čeznju in jih paroma zlagamo v novi seznam. Če pri tem uporabimo zip, bo ta sam odrezal odvečne elemente.

Spremeniti moramo podani seznam. Tega ne moremo storiti s prirejanjem nekega novega seznama imenu argumenta (v slogu `s = nekaj`), temveč moramo dejansko spreminjati podani seznam: najprej ga bomo izpraznili s `clear`, nato pa vanj zložili, kar je treba.

```
def soda_liha(s):
    soda = [x for x in s if x % 2 == 0]
    liha = [x for x in s if x % 2 == 1]
    s.clear()
    for so, li in zip(soda, liha):
        s += (so, li)
```

74. Plus – minus – plus

Najpreprosteje je števila spravljati v nov seznam in na koncu zamenjati vsebino originalnega seznama z novo.

```
def alterniraj(x):
    n = []
    for e in x:
        if not n or e * n[-1] < 0:
            n.append(e)
    x[:] = n
```

Glavna umetnost se skriva v pogoju v `if`, ki odloča, ali bomo nek element `e` dodali v novi seznam ali ne. Tako pravi: dodati ga je potrebno, če je novi seznam prazen (`not n`), ali pa imata element in zadnji element seznama `n` (`e` in `n[-1]`) različen predznak. To bi lahko preverjali z `e < 0 and n[-1] > 0 or e > 0 and n[-1] < 0`, vendar gre preprosteje: pomnožimo ju in če je predznak različen, bo produkt negativen.

Funkcija se mora končati z `x[:] = n` in ne `x = n`. Prvo zamenja vse elemente seznama x z novimi. Torej: spreminja seznam x. Drugo priredi ime `x` seznamu, ki se imenuje (tudi) `n`. Seznam, na katerega se je pred tem nanašalo ime `x`, pa je ostal nespremenjen.

Z nekaj previdnosti pa lahko spreminjamo x kar sproti, ne da bi sestavljali nov seznam.

```
def alterniraj(x):
    ostane = 1
    for e in x:
        if e * x[ostane - 1] < 0:
            x[ostane] = e
            ostane += 1
    del x[ostane:]
```

Elementov, ki jih bomo odstranili, ne smemo brisati sproti: znotraj zanke `for` nikoli ne smemo spreminjati seznama, po katerem zanka teče. Pač pa na začetek seznama prestavljamo elemente, ki jih nameravamo ohraniti.

Spremenljivka `ostane` bo štela elemente, ki jih nameravamo ohraniti. Za začetek bomo ohranili vsaj enega, prvega. Nato gremo čez seznam. Če ima določen element e drugačen predznak od zadnjega, ki ga bomo ohranili (ta je na mesto `x[ostane - 1]`, saj `ostane` šteje ohranjene elemente, indeksi pa se začnejo z `0`), shranimo ta element med ohranjene in povečamo `ostane`. Po zanki pobrišemo iz seznama `x` vse elemente, ki ne bodo ostali v njem. Točneje, pobrišemo odvečni del seznama.

Bi morali zanko napisati kot `for e in x[1:]:` in ne `for e in x:`, saj smo se za prvega že odločili, da bo ohranjen? Lahko bi, vendar ni potrebe. V prvem koraku bomo primerjali prvi element samega s seboj in ker bo imel enak predznak, ga bomo preskočili.

75. Pecivo

Primerjati moramo zaporedne pare črk. Ko vidimo A, preverimo, kaj je pred njim. Če O, med dvema "paketoma" niso prodali ničesar. Če B, so prodali en kos. Če A, dva.

Zaporedne pare najlepše primerjamo z `zip(s, s[1:])`. Na konec pa prilepimo še en `A`, da bomo v primeru, da se zaporedje konča z, na primer, `...AB` prišteli prodani `O`.

```
def pecivo(s):
    vseh = 0
    for prej, ta in zip(s, s[1:] + "A"):
        if ta == "A":
            if prej == "A":
                vseh += 2
            elif prej == "B":
                vseh += 1
    return vseh
```

Gre pa tudi hitreje, če se domislimo, da so spekli (vsaj) trikrat toliko kosov peciva, kot je A-jev. Prodali pa so toliko, kolikor ga ni v nizu. Torej

```
def pecivo(s):
    return 3 * s.count("A") - len(s)
```

Ha ha.

76. Nogavice brez para

S številkami ravnajmo enako, kot bi z nogavicami. Zlagajmo jih na mizo (oziroma v nov seznam). Za vsako novo nogavico najprej pogledamo, ali je ena taka že na mizi. Če je, jo odstranimo. Če ni, jo dodamo.

```
def nogavice(s):
    brez_para = []
    for e in s:
        if e in brez_para:
            brez_para.remove(e)
        else:
            brez_para.append(e)
    return brez_para
```

Z nekaj spretnosti – še krajše:

```
def nogavice(s):
    brez_para = []
    for e in s:
        [brez_para.append, brez_para.remove][e in brez_para](e)
    return brez_para
```

Metodi `brez_para.append` in `brez_para.remove` zložimo v seznam. Pogoj `e in brez para` bo `False` ali `True`, kar je isto kot 0 ali 1. Uporabimo ga kot indeks; če bo pogoj enak `False` (nogavica je brez para) bo vzel ničti element seznama `[brez_para.append, brez_para.remove]`. Če bo `True`, bo izbral prvega. Rezultat izraza `[brez_para.append, brez_para.remove][e in brez_para]` je torej `brez_para.append` ali `brez_para.remove`. Ta rezultat pokličemo z argumentom `e`.

77. Presedanje

Dva razporeda sta enaka, če enega lahko "preobrnemo" v drugega, tako, da postavimo vse goste od `i`-tega naprej na začetek. To nam da skoraj pravilno rešitev naloge:

```
def enaka_razporeda(razpored1, razpored2):
    for i in range(len(razpored2)):
        if razpored2 == razpored1[i:] + razpored1[:i]:
            return True
    return False
```

Števec `i` pove, za koliko mest zasukamo prvi razpored. Čim obstaja `i`, zasuk, ki da enak razpored, vrnemo `True`. Če ga ni, na koncu vrnemo `False`.

Težave imamo le še, če sta obe mizi prazni. V tem primeru sta namreč enaki, vendar ne uspemo priti do `return True`, saj se zanka nikoli ne izvede. To lahko rešimo tako, da na začetek funkcije damo

```
    if not razpored1 and not razpored2:
        return True
```

Lahko pa rečemo celo kar

```
if razpored1 == razpored2:
    return True
```

To poskrbi tako za primer, ko sta obe mizi prazni, kot za primer, ko gostje sedijo popolnoma enako. Zanko for lahko tedaj spremenimo tako, da ne gre prek `range(len(razpored2))`, temveč zadošča `range(len(razpored2))`, saj smo za primer, ko sta razporeda enaka brez vrtenja, že poskrbeli.

Če hočemo krajši program, pa to zrinemo v `return`: četudi znotraj zanke nismo vrnili `True`, je pravilni odgovor še vedno `True`, če sta oba seznama prazna.

```
def enaka_razporeda(razpored1, razpored2):
    for i in range(len(razpored2)):
        if razpored2 == razpored1[i:] + razpored1[:i]:
            return True
    return not razpored1 and not razpored2
```

Če ni nobeden od seznamov prazen, bo `not razpored1 and not razpored2` pač `False`, tako kot mora biti.

Nekoliko daljša, a hitrejša rešitev ne uporablja zanke (vsaj mi je ne napišemo), ker ugane, koliko je treba zavrteti mizo. To stori tako, da pogleda, kje v drugem razporedu sedi prva oseba iz prvega razporeda.

```
def enaka_razporeda(razpored1, razpored2):
    prvi = razpored1[0]
    kjePrvi = razpored2.index(prvi)
    return razpored1 == razpored2[kjePrvi:] + razpored2[:kjePrvi]
```

Tu smo, kot dovoljuje naloga, predpostavili, da imajo osebe različna imena. Če vzamemo prvo osebo iz enega razporeda in pogledamo, kje se le-ta pojavi v drugem razporedu, že vemo, za koliko jo je potrebno zavrteti mizo. Če je prvi razpored enak ustrezno zavrtenemu drugemu, sta razporeda enaka, sicer ne.

Rešiti moramo le še dve nevšečnosti. Lahko se zgodi, da je prvi razpored prazen in v tem primeru ne moremo dobiti njegovega prvega elementa, `prvi = razpored1[0]`. Če je prvi razpored prazen, sta razporeda enaka natanko takrat, ko je prazen tudi drugi razpored. Druga nesreča nas čaka takoj v naslednji vrsti: lahko se zgodi, da prve osebe prvega razporeda sploh ni v drugem razporedu. V tem primeru razporeda nista enaka.

Za obe izjemi poskrbimo z dodatnimi pogoji.

```
def enaka_razporeda(razpored1, razpored2):
    if not razpored1:
        return not razpored2
    prvi = razpored1[0]
    if not prvi in razpored2:
        return False
    kjePrvi = razpored2.index(prvi)
    return razpored1 == razpored2[kjePrvi:] + razpored2[:kjePrvi]
```

Slovarji in množice

78. Pokaži črke

Najprej sestavimo prazen niz `bes`. Nato gremo čez dano besedo, črko za črko. Če je določena črka v množici, v `bes` dodamo to črko, sicer dodamo namesto nje samo piko.

```
def pokazi_crke(beseda, crke):
    bes = ""
    for c in beseda:
        if c in crke:
            bes += c
        else:
            bes += "."
    return bes
```

~~Če znamo uporabiti `if-else`~~ Ker znamo uporabiti `if-else`, lahko napišemo takole.

```
def pokazi_crke(beseda, crke):
    bes = ""
    for c in beseda:
        bes += c if c in crke else "."
    return bes
```

Krajša rešitev, ki uporablja izpeljani seznam, je,

```
def pokazi_crke(beseda, crke):
    return "".join((c if c in crke else ".") for c in beseda)
```

79. Podobna beseda

Tokrat v imenu poučnosti začnimo z napačno rešitvijo

```
# Ta funkcija ne deluje pravino!
def podobna(beseda):
    beseda = beseda.lower()
    naj_crk = -1
    for ena_beseda in besede:
        ta = 0
        for c in beseda:
            if c in ena_beseda:
                ta += 1
        if ta > naj_crk:
            naj_beseda = ena_beseda
            naj_crk = ta
    return naj_beseda
```

Težava je v tem, da enake črke prve besede štejemo večkrat. Ana in krava bi imeli štiri skupne črke, saj se dva Anina a-ja ujemata z dvema kravjima.

V takih primerih se lahko vedno zatečemo k množicam: te vsebujejo vsako reč le enkrat. Število skupnih črk je kar enako preseku (dobimo ga z operatorjem `&`) množice črk iz ene in iz druge besede.

```
def podobna(beseda):
    beseda = beseda.lower()
    bs = set(beseda)
    naj_crk = -1
    for ena_beseda in besede:
        skupnih = len(bs & set(ena_beseda))
        if skupnih > naj_crk:
            naj_crk = skupnih
            naj_beseda = ena_beseda
    return naj_beseda
```

Rešitve brez množic so veliko dolgoveznejše. Napišimo eno.

```
def podobna(beseda):
    beseda = beseda.lower()
    naj_crk = -1
    for ena_beseda in besede:
        skupne = []
        for crka in beseda:
            if crka in ena_beseda and not crka in skupne:
                skupne.append(crka)
        skupnih = len(skupne)
        if skupnih > naj_crk:
            naj_crk = skupnih
            naj_beseda = ena_beseda
    return naj_beseda
```

Seznam `skupne` v bistvu predstavlja presek: v njem so vse črke iz besede (`for crka in beseda`), ki so tudi v drugi besedi (`crka in ena_beseda`). Z `and not crka in skupne` poskrbimo, da dodamo vsako črko le enkrat.

Rokohitreci vedo povedati, da ima funkcija `max` argument `key`, s katerim lahko nalogo rešijo kar takole.

```
def podobna(beseda):
    return max(besede, key=lambda x: len(set(beseda.lower()) & set(x)))
```

80. Število znakov

Spet najprej poglejmo nerodno rešitev: z zanko gremo čez črke besede in v seznam zlagamo vse črke, ki še niso v seznamu. To storimo za vsako besedo in vrnemo tisto z najdaljšim seznamom.

```
def najraznolika(bes):
    naj_crk = 0
    for b in bes:
        crke = []
        for c in b.lower():
            if not c in crke:
                crke.append(c)
        if len(crke) > naj_crk:
            naj_crk = len(crke)
            naj_beseda = b
    return naj_beseda
```

Spet je veliko elegantnejše je taisto narediti z množicami.

```
def najraznolika(bes):
    naj_crk = 0
    for b in bes:
        crk = len(set(b.lower()))
        if crk > naj_crk:
            naj_crk = crk
            naj_beseda = b
    return naj_beseda
```

Rokohitreci se spet spomnijo, da ima funkcija `max` argument `key` in nalogo rešijo v enem zamahu.

```
def najraznolika(bes):
    return max(bes, key=lambda x:len(set(x.lower())))
```

81. Najpogostejša beseda in črka

Začnimo s pomožno funkcijo, ki izračuna pogostosti elementov v podanem seznamu; vrne jih v obliki slovarja.

```
def frekvence(xs):
    f = {}
    for x in xs:
        if not x in f:
            f[x] = 1
        else:
            f[x] += 1
    return f
```

Za vsak x iz seznama preverimo, ali je v slovarju `f`. Če ga ni, povemo, da se je pojavil enkrat. Če je, povemo, da se je pojavil še enkrat. Ker podobno reč delamo res velikokrat, ponuja

Pythonov modul `collections` slovar s privzetimi vrednostmi. Če `f` še nima ključa `x`, se bo v slovarju pojavil sam od sebe, njegova začetna vrednost pa bo `0` (a jo bomo takoj povečali za 1).

```
import collections
def frekvence(xs):
    f = collections.defaultdict(int)
    for x in xs:
        f[x] += 1
    return f
```

Več na temo privzetih vrednosti v slovarjih bomo napisali v rešitvi naloge Družinsko drevo. Od Pythona 3.0 nam je za takale preštevanja na voljo prikladen razred Counter, ki se prav tako nahaja v modulu collections in dela pravzaprav prav isto kot gornja funkcija: namesto

```
f = frekvence(s)
```

lahko napišemo preprosto

```
f = collections.Counter(s)
```

pa smo.

Zdaj napišimo funkcijo, ki preišče takšenle slovar, dobljen na prvi ali drugi način, in vrne ključ elementa z največjo vrednostjo (se pravi, besedo z največjim številom ponovitev).

```
def najpogostejsi(f):
    naj_vred, naj_ponov = "", 0
    for vred, ponov in f.items():
        if ponov > naj_ponov:
            naj_vred, naj_ponov = vred, ponov
    return naj_vred
```

Tudi tole gre krajše, če znamo modro uporabiti vgrajeno funkcijo `max` (razlagali spet ne bomo; naj razume, kdor ve dovolj).

```
def najpogostejsi(f):
    return max(f.items(), key=lambda x:x[1])[0]
```

Zdaj, ko smo kakorsižebodi pripravili pomožni funkciji, se lotimo iskanja najpogostejših besed in črk.

```
def najpogostejsa_beseda(s):
    besede = s.split()
    pogostosti = frekvence(besede)
    najpogostejsa = najpogostejsi(pogostosti)
    return najpogostejsa
```

Funkcijo smo napisali karseda nazorno, korak za korakom. V resnici takole programirajo kvečjemu najpotrpežljivejši budistični menihi. Po krščansko se taista reč napiše

```
def najpogostejsa_beseda(s):
    return najpogostejsi(frekvence(s.split()))
```

Pa najpogostejša črka? Ko smo pisali funkcijo frekvence, smo si predstavljali, da bo kot argument dobila seznam; zanka `for` gre prek elementov seznama. Vendar sme kot argument

dobiti tudi niz; zanka bo šla v tem primeru prek črk niza in funkcija bo namesto pogostosti elementov v seznamu vrnila pogostosti črk v nizu. Torej,

```
def najpogostejsi_znak(s):
    return najpogostejsi(frekvence(s))
```

In amen.

82. Samo enkrat

Naloga je tako preprosta, da je ne bomo rešili le enkrat, temveč trikrat.

Najprej najpreprostejša rešitev – ki je tudi zaslužna, da se je naloga znašla v sklopu nalog o množicah. Niz pretvorimo v množico. Ta bo imela vsak znak samo po enkrat. Če je velikost množice enaka dolžini niza, je imel tudi niz vsak znak samo po enkrat.

```
def samo_enkrat(s):
    return len(set(s)) == len(s)
```

Seveda vrnemo rezultat primerjave. Ta je že `True` ali `False`. Kdor misli, da je

```
def samo_enkrat(s):
    if len(set(s)) == len(s):
        return True
    else:
        return False
```

spodobna rešitev, živi v hudi zmoti.

Kot vedno pa obstajajo tudi daljše rešitve. Lahko gremo prek niza in za vsako črko preverimo, ali se je že pojavila: če se i-ta črka že pojavi med prvimi i-timi, se očitno pojavi (vsaj) dvakrat.

```
def samo_enkrat(s):
    for i, c in enumerate(s):
        if c in s[:i]:
            return False
    return True
```

Grobijani pa nalogo rešijo s preštevanjem: za vsako črko preštejejo, kolikokrat se pojavi in če se pojavi več kot enkrat, je veselja konec.

```
def samo_enkrat(s):
    for c in s:
        if s.count(c) > 1:
            return False
    return True
```

83. Popularni rojstni dnevi

Naloga je čista klasika, le da navadno štejemo pogostosti besed v besedilu, kot v nalogi Najpogostejša beseda in črka. Tu pač štejemo datume, ostalo je podobno.

Pa jo zato tokrat rešimo nekoliko drugače. Pripravimo seznam, v katerem bo za vsak datum pisalo, koliko ljudi ima tedaj rojstni dan. V, na primer, `dnevi[405]` bo pisalo, koliko dni ima rojstni dan petega aprila (`405` bo pomenilo "april peti").

```
def histogram_dni(imedat):
    dnevi = [0] * 1232
    for l in open(imedat):
        dnevi[int(l[2:4] + l[:2])] += 1
    for d, c in enumerate(dnevi):
        if c > 0:
            print("%2i. %2i. %i" % (d % 100, d // 100, c))
```

V tabeli bodo tudi elementi 0 (0.-ti dan 0-tega meseca), pa 10 (kar ustreza 0010, deseti dan ničtega meseca). Elementi z nesmiselnimi indeksi nas ne bodo vznemirjali, saj bodo prazni in jih tudi izpisali ne bomo. Paziti moramo le, da pripravimo dovolj veliko tabelo; 366 dni, kolikor je različnih datumov, očitno ne bo dovolj. Največji indeks, ki se bo še pojavil, je 31. december, ki ga bomo zapisali v `1231`. Tabela, katere zadnji element ima indeks `1231`, ima `1232` elementov. Toliko jih pripravimo; 0-ti mesec in 35. maj bosta pač odveč.

Prva zanka `for` gre prek datoteke in prešteva rojstne dni tako, da povečuje števce v ustreznih predalčkih tabele. Izraz `l[2:4] + l[:2]` zamenja dneve in mesece: EMŠO se začne z dnevom in nadaljuje z mesecem, mi pa bi radi obratno, da bodo stvari pravilno urejene. Ko torej naletimo na nekoga, ki je rojen petega aprila, poveča `dnevi[405]` za 1.

Druga zanka izpisuje. Uporabili smo `enumerate`, da bo v `d` indeks elementa (npr. `405` za 5. april), v `c` pa števec. Datum izpišemo le, če ima kdo na ta dan rojstni dan. Dan dobimo kot ostanek po deljenju s 100 (`405 % 100=5`), mesec pa s celoštevilskim deljenjem s 100 (`405 // 100=4`). Kdor je vajen starejših različic Pythona, bi morda napisal `405 / 100`; v Pythonu 3.0 in kasnejših bi to dalo 4.05 in ne 4.

Je rešitev, ki smo jo napisali, boljša ali slabša od štetja pogostosti s slovarjem, ki smo ga uporabili, recimo, v nalogi Najpogostejša črka? Popoln odgovor bodo dali predmeti s področja podatkovnih struktur, nakažemo pa ga že lahko. Navidez je naša rešitev potratnejša, saj uporabljamo seznam s 1232 elementi, v slovarju pa bi bili le tisti datumi, ki se v resnici pojavljajo. Slovar bi imel v najslabšem primeru 366 elementov, seznam jih ima več kot trikrat toliko. V resnici pa nas en element slovarja stane več kot en element seznama. Slovarji in množice so narejeni tako, da lahko Python hitro odkrije, ali vsebujejo določen ključ, za kar uporabljajo takoimenovano razpršeno tabelo. Ta zahteva veliko pomnilnika. Seznam, v katerem je deset milijonkrat napisano `True`, zasede okrog 40 megabajtov. Slovar z enakim številom elementov zasede devetkrat toliko. Opisana rešitev naloge najbrž porabi manj pomnilnika od rešitve s slovarji. To pa seveda ne pomeni, da se je potrebno slovarjev bati. Slovarji in množice so vseeno zakon; navdušeno jih uporabljajte, kjerkoli vam pridejo prav.

84. Osrednja obveščevalna služba

Najprej rešimo nalogo nekoliko narobe. Pripravimo seznam besed in potem za vsako besedo v seznamu izpišimo, kolikokrat se pojavi.

```
# Ne deluje povsem pravilno
seznam = msg.split()
for beseda in seznam:
    if beseda.istitle():
        print(beseda, seznam.count(beseda))
```

Vsako ime se izpiše večkrat – pač tolikokrat, kolikorkrat se pojavi. Tega se lahko znebimo tako, da si zapomnimo, kaj smo že izpisali.

```
izpisano = set()
seznam = msg.split()
for beseda in seznam:
    if beseda.istitle() and  not beseda in izpisano:
            print(beseda, seznam.count(beseda))
            izpisano.add(beseda)
```

Nekoliko moteče je stalno preštevanje: za vsako besedo iz seznama, ki predstavlja ime, bo `count` (ponovno) preletel vse besede seznama. Boljše bi bilo, če bi program napisali brez `count`, tako, da bi preštevali sami. Število pojavitev besede bomo shranjevali v slovar z imenom `pojavitve`. Nalogo smo s tem spet prevedli na staro dobro klasiko preštevanja pojavitev.

```
import collections
pojavitve = collections.defaultdict(int)
seznam = msg.split()
for beseda in seznam:
    if beseda.istitle():
        pojavitve[beseda] += 1
for beseda, stevec in pojavitve.items():
    print(beseda, stevec)
```

Tudi pri tej nalogi bi si sicer lahko pomagali s `collections.Counter`. Ker so takšne reči le za tiste, ki že kaj znajo (mlajši naj si raje brusijo svoje mlečne zobe s premetavanjem zank in pogojev), jo nakažimo le v eni vrstici

```
collections.Counter(word for word in msg.split() if word.istitle())
```

Malenkost, ki manjka, naj dopiše vsak sam.

85. Menjave

Čeprav obstajajo tudi precej daljše rešitve :), pokažimo kratko.

```
def zamenjano(skatla, menjave):
    return [menjave.get(e, e) for e in skatla]

def zamenjaj(skatla, menjave):
    skatla[:] = zamenjano(skatla, menjave)
```

Slovarji, vemo, imajo metodo `get`, ki vrne vrednost, ki pripada podanemu ključu. Če ključ ne obstaja, pa vrnejo podano privzeto vrednost. Tako `menjave.get(e, e)` vrne vrednost, ki pripada ključu `e`, sicer pa kar `e`. To zložimo v seznam, ki ga vrnemo: vsak element zamenjamo, s tistim, s čimer ga je potrebno zamenjati, ali pa ga ne zamenjamo.

Funkcija `zamenjaj` pa zamenja vse elemente škatle s tem, kar vrne funkcija `zamenjano`.

86. Anagrami

Pri mnogih nalogah – od Razcep do Drugi največji element – se je program precej ujemal z načinom, na katerega bi nalogo reševali ročno. Tule ta metoda vodi v ne preveč všečno rešitev. "Ročno" bi nalogo reševali tako, da bi gledali črke ene besede in (v mislih) črtali črke druge.

```
def anagram(b1, b2):
    crke2 = list(b2)
    for crka in b1:
        if not crka in crke2:
            return False
        else:
            crke2.remove(crka)
    return crke2 == []
```

Besedi sta anagrama, če smo uspeli v `crke2` najti vsako črko iz `crke1` (s tem, da črke iz `crke2` vmes brišemo) in če so na koncu vse črke druge besede prečrtane. Konec smo napisali nekoliko jasneje, kot bi ga v praksi; v resnici bi namesto primerjanja s praznim seznamom napisali `return not crke2`.

Črke druge besede smo spremenili v seznam, ker je brisanje elementov seznama "naravnejša" operacija kot brisanje črk niza. Seveda pa lahko poskusimo tudi z nizi.

```
def anagram(b1, b2):
    if len(b1) != len(b2):
        return False
    for crka in b1:
        b2 = b2.replace(crka, "", 1)
    return not b2
```

Če sta niza različnih dolžin, nista anagrama. Sicer gremo prek črk prve besede in zamenjamo dotično črko v drugi besedi s praznim nizom; z drugimi besedami, pobrišemo jo. Tretji argument v `replace`, `1`, pomeni, da želimo pobrisati največ eno ponovitev. Če te črke v drugem nizu ni (več), je pač ne bomo pobrisali. Ker v drugem nizu pobrišemo (največ) toliko črk, kolikor jih ima prvi in ker sta v začetku oba niza enako dolga, je lahko drugi niz na koncu prazen le, če smo vsakič pobrisali po eno črko. To pa se zgodi natanko takrat, ko so v drugem nizu iste črke kot v prvem.

S prvo rešitvijo, seznami, ni nič narobe, vendar zasluži 0 točk za eleganco. Druga rešitev je vsaj duhovita, vendar tudi dokaj neučinkovita in grda, ker v njej brišemo črke iz niza. To se ne dela (brez dobrega vzroka).

Če je ob iskanju boljše rešitve kdo pomislil na množice, naj pomisli še enkrat. Besedi "roka" in "korak" nista anagrama, saj ima korak en "k" več, množicam pa bi se zdelo, da sta, saj imata ista črke. Vendar ideja ni povsem napačna.

Besedi sta anagrama, če se v njima vse črke pojavijo enakokrat. Spomnimo se torej, kaj smo počeli v nalogi Najpogostejša beseda in črka in napišimo kratko in jedrnato

```
def anagram(b1, b2):
    return collections.Counter(b1) == collections.Counter(b2)
```

Rešitev, ki bi jo napisal vsak programer v Pythonu, ki ima kaj samospoštovanja, pa je takšna: črke besede uredimo po abecedi.

```
>>> sorted("pirat")
['a', 'i', 'p', 'r', 't']
```

Če dobimo pri obeh besedah isto, gre za anagram.

```
def anagram(b1, b2):
    return sorted(b1) == sorted(b2)
```

Ob Popularnih rojstnih dnevih smo si dovolili nekaj teoretičnih misli o podatkovnih strukturah. Še tu povejmo nekaj malega, le toliko, da boste zaslutili, da so tu zadaj še veliko globje reči. Vzemimo, da so besede "kar dolge". Če jih še podaljšujemo, se s tem podaljšuje tudi čas, ki ga porabi funkcija, vendar pri zadnjih dveh rešitvah čas, ki ga potrebuje funkcija, narašča veliko počasneje kot pri prvih dveh, z brisanjem. Prvi dve za dvakrat daljšo besedo potrebujeta kar štirikrat več časa. Po domače: če so besede zelo dolge, sta zadnji rešitvi precej hitrejši. Če so kratke, je čas zanemarljiv pri vseh različicah; v tem primeru nam je spet najbolj všeč zadnja rešitev, ker je funkcija najkrajša, celo tako kratka, da se v njej nimamo kje zmotiti. (Da, prednost kratkih programov je tudi v tem, da imajo manj napak!) Rešitev s pogostostmi je teoretično najhitrejša – odvisno od detajlov tega, kako so narejeni Pythonovi slovarji –, vendar za to žrtvuje dodatni pomnilnik.

87. Bomboniera

Naloga je umeščena med naloge na temo množic. Upamo, da je to koga, ki bi se je lotil na kak drug, nerodnejši način, spodbudilo, da je razmislil, kako jo rešiti z množicami.

Naloga je zelo preprosta, če razmislimo, kaj pravzaprav hoče: število bombonov, ki jih bo snedel Benjamin, je enaka produktu števila nedotaknjenih vrstic in nedotaknjenih stolpcev. Prešteti je torej potrebno "dotaknjene" vrstice in stolpce. Vsako vrstico in stolpec seveda štejemo le enkrat. Torej množice.

```
    stolpci = set()
    vrstice = set()
    for s, v in pojedeno:
        stolpci.add(s)
        vrstice.add(s)
```

ali, krajše (če že poznamo generatorje)

```
stolpci = {s for s, v in pojedeno}
vrstice = {v for s, v in pojedeno}
```

Četudi je iz istega stolpca (ali vrstice) morda vzela več bombonov, se v množici pojavi le enkrat - ker je to pač množica. Število dotaknjenih stolpcev in vrstic je potem `len(stolpci)` in `len(vrstice)`, število nedotaknjenih pa `sirina - len(stolpci)` in `visina - len(vrstice)`.

Pa smo rešili nalogo.

```
def bomboniera(sirina, visina, pojedeni):
    stolpci = set()
    vrstice = set()
    for s, v in pojedeno:
        stolpci.add(s)
        vrstice.add(s)
    return (sirina – len(stolpci)) * (visina – len(vrstice))
```

ali, krajše,

```
def bomboniera(sirina, visina, pojedeni):
    return (sirina - len({x for x, _ in pojedeni})) \
         * (visina - len({y for _, y in pojedeni}))
```

88. Prafaktorji in delitelji

Prvi, težji del naloge že poznamo (glej nalogo Razcep na prafaktorje).

```
def prafaktorji(n):
    fact = {}
    for i in range(2, n + 1):
        for d in range(n):
            if n % i == 0:
                n //= i
            else:
                break
        if d != 0:
            fact[i] = d
    return fact
```

Z zunanjo zanko pošljemo `i` prek vseh števil od `2` do `n`. Nato z notranjo zanko preštejemo, kolikokrat `i` deli `n`: z `d`-jem štejemo od `0` do `n`. Če `i` deli `n`, ga (celoštevilsko, z `//`) delimo. Sicer prekinemo zanko. Notranja zanka se izvede nobenkrat, enkrat, dvakrat ... tolikokrat, kolikorkrat je pač mogoče deliti `n` z `i`. Po zanki preverimo, ali smo `n` kdaj delili z `i` in če smo ga, to zapišemo.

Funkcija je počasna: za `i` ni potrebno, da gre čisto do `n`. Pospešimo jo lahko na različne načine. Vstavimo lahko, recimo,

```
if i > n:
    break
```

ali, kar se izkaže za isto

```
if n == 1:
    break
```

med prvi in drugi `for`.

Funkcija bo še vedno kar počasna, a razcep na praštevila pač ni hiter posel.

Kako lahko vemo, da bodo delitelji praštevila? Kaj, če bi bil `i`, recimo `6`? V tem primeru ne bo delil `n`-ja. Preden bo `i` priplezal do `6`, bo enak `2` in `3`; ko `n` ni več deljiv ne z `2` ne s `3`, tudi s `6` ne more biti.

Zanimiva druga rešitev je tale:

```
def prafaktorji(n):
    fact = defaultdict(int)
    i = 2
    while n > 1:
        if n % i == 0:
            fact[i] += 1
            n //= i
        else:
            i += 1
    return fact
```

Če `n` deli `i`, povečamo potenco `fact[i]` za `1` in delimo `n`. Sicer gremo na naslednji `i`.

Funkcija `gcd` dobi dva slovarja. Zanimajo nas ključi, ki se pojavijo v obeh; iz slovarjev vzamemo tisto vrednost pri tem ključu, ki je manjša. Vse skupaj množimo v končno število `n`: če je v enem slovarju `7:3` in v drugem `7:2`, bomo `n` pomnožili s `7:2`.

```
def gcd(f1, f2):
    n = 1
    for p in f1:
        if p in f2:
            n *= p ** min(f1[p], f2[p])
    return n
```

To se da skrajšati na različne načine, recimo tako, da uporabimo `get`: če drugi slovar nima tega ključa, se delajmo, da obstaja, vendar ima potenco `0`. `p ** 0` bo `1`, in `n *= p ** 0` ne bo spremenil `n`-ja.

```
def gcd(f1, f2):
    n = 1
    for p, m1 in f1.items():
        m2 = f2.get(p, 0)
        n *= p ** min(m1, m2)
    return n
```

Še elegantneje gre z množicami: izračunamo presek ključev. Potem gremo prek tega preseka in se tako znebimo pogojev oziroma `get`-a. Ostanek je podoben kot prej

```
def gcd(f1, f2):
    n = 1
    for p in set(f1) & set(f2):
        n *= p ** min(f1[p], f2[p])
    return n
```

89. Hamilkon

Naloga zahteva, da pregledamo vse, kar bi lahko bilo narobe in vrnemo `False`, čim najdemo kaj takega. Če ne najdemo ničesar, bomo vrnili `True`. Prva pogoja sta očitna: obhod je dolg 65 polj (šahovnica jih ima 64, vendar se prvo in zadnje ponovi) in prvo polje mora biti enako zadnjemu.

```
if len(obhod) != 65 or obhod[0] != obhod[-1]:
    return False
```

Vrstni red teh dveh pogojev je pomemben: če funkcija dobi prazen seznam, se bo računanje pogoja končalo že ob prvem. Če bi ju obrnili, bi najprej preverjali `obhod[0]` in program bi vrnil napako, saj prazen seznam nima ničtega elementa.

Zdaj bomo preverili ali se vsako polje (razen prvega) pojavi le enkrat. Neroden programer bi za vsako polje (razen prvega) preštel, kolikokrat se pojavi, in zarohnel, če več kot enkrat.

```
for polje in obhod[1:]:
    if obhod.count(polje) > 1:
        return False
```

Ker se ravno ukvarjamo z množicami, znamo to narediti boljše: če spravimo vseh 65 polj v množico, mora imeti 64 elementov, saj se ponovi le eno polje. Katero, pa tudi vemo, saj smo že preverili – prvo in zadnje.

Zdaj pa še za vsak skok preverimo, ali je pravilen. Recimo, da sta `p1` in `p2` polji, med katerima je skočil konj. S funkcijo `ord`, že vemo, dobimo ASCII kodo znaka, torej `ord(p2[0]) - ord(p1[0])` pove, koliko stolpcev in `ord(p2[1]) - ord(p1[1])`, koliko vrstic daleč je skočil. Ker nam ni potrebno ločevati med levo in desno ter gor in dol, vzamemo kar absolutne vrednosti; hočemo, da je absolutna vrednost ene od teh dveh razlik 1, druga pa 2. V vsakem jeziku lahko napišemo

```
if not (abs(ord(p2[0]) - ord(p1[0])) == 1 and abs(ord(p2[0]) - ord(p1[0])) == 2 or
        abs(ord(p2[0]) - ord(p1[0])) == 2 and abs(ord(p2[0]) - ord(p1[0])) == 1):
    return False
```

V Pythonu lahko obe razdalji postavimo v množico in ta množica mora biti enaka `{1, 2}`. Tako dobimo tole elegantno rešitev naloge.

```
def hamilkon(obhod):
  if len(obhod) != 65 or obhod[0] != obhod[-1] or len(set(obhod)) != 64:
      return False
  for i in range(64):
     p1, p2 = obhod[i], obhod[i+1]
     if {abs(ord(p1[0]) - ord(p2[0])), abs(ord(p1[1]) - ord(p2[1]))} != {1, 2}:
         return False
  return True
```

Ker preverjamo relacijo med zaporednimi elementi seznama, gremo z nekim števcem, tu smo ga imenovali `i`, od 0 do 63 ter preverjamo polji `obhod[i]` in `obhod[i+1]`. Če ima kdo s tem težave, naj ponovno reši sorodno nalogo Nepadajoči seznam.

90. Družinsko drevo

Reševanje naloge bomo izkoristili kot priložnost, da povadimo delo s slovarji s privzetimi vrednostmi.

Najprej rešimo nalogo z običajnim slovarjem. Odpreti moramo datoteko, jo brati po vrsticah in vsako vrstico razdeliti na imeni. Če imena starša še ni v slovarju, ga dodamo, kot ključ podamo prazen seznam. Nato v seznam, ki pripada staršu (stari ali pravkar dodani) dodamo otroka.

```
def beri_drevo(ime_dat):
    drevo = {}
    for v in open(ime_dat):
        stars, otrok = v.split()
        if not stars in drevo:
            drevo[stars] = []
        drevo[stars].append(otrok)
    return drevo
```

Preverjanje, ali starša že poznamo, je nadležno. Rešimo se ga lahko z uporabo slovarjeve metode `setdefault`, ki vrne vrednost s podanim ključem, če le-ta obstaja; sicer pa doda takšno vrednost v slovar (in jo vrne).

```
def beri_drevo(ime_dat):
    drevo = {}
    for v in open(ime_dat):
        stars, otrok = v.split()
        drevo.setdefault(stars, []).append(otrok)
    return drevo
```

V resnici dela ta program natanko tako kot prvi, le `if` je namesto nas izvedla metoda `setdefault`. Pa malenkost hitreje deluje. Tudi, da je krajši, nam je ljubo; da manj pregleden, pa ne. Zato uporabimo glavni trik: v Pythonovem modulu `collections` se skrivajo slovarji s privzetimi vrednostmi.

```
import collections
def beri_drevo(ime_dat):
    drevo = collections.defaultdict(list)
    for v in open(ime_dat):
        stars, otrok = v.split()
        drevo[stars].append(otrok)
    return dict(drevo)
```

S `collections.defaultdict(list)` povemo, naj se v slovarju sami od sebe pojavljajo novi elementi, njihove vrednosti pa naj bodo prazni seznami (tisto, kar dobimo, če pokličemo "funkcijo" `list`). V programu zdaj mirno dodajamo otroke staršem, ne da bi se vznemirjali, ali so se starši že kdaj pojavili ali ne.

Na koncu pretvorimo slovar s privzetimi vrednostmi v običajen slovar, saj tako zahteva naloga, poleg tega pa se bomo s tem rešili težav v prihodnosti (boste videli, katerih).

Funkcija, ki vrne otroke podanega starša, je skoraj trivialna.

```
def otroci(drevo, ime):
    return drevo[ime]
```

Kaj pa, če oseba nima otrok? V tem primeru bi bilo primerno vrniti prazen seznam. Pravzaprav bi se to, če bi ostali pri slovarju s privzetimi vrednostmi, že zgodilo: ob `drevo[ime]` bi se v slovar dodal nov element s ključem ime in praznim seznamom kot vrednostjo. Vendar tega nočemo: funkcija `otroci` naj ne bi dodajala v drevo, temveč le poročala, kaj je v njem. Z eno ali drugo obliko slovarja, boljša funkcija bi bila takšna:

```
def otroci(drevo, ime):
    if ime in drevo:
        return drevo[ime]
    else:
        return []
```

Spet nam ni všeč tole `if`-anje. Tudi krajša različica,

```
def otroci(drevo, ime):
    return drevo[ime] if ime in drevo else []
```

ne poteši povsem našega izpiljenega čuta za estetiko. Rešitev, ki bi jo uporabil zaresen programer (če bi se zaresni programerji ukvarjali s takimi otročarijami), je

```
def otroci(drevo, ime):
    return drevo.get(ime, [])
```

Metoda `get` vrne, kar je v slovarju, če tega ne najde, pa vrne, kar podamo kot drugi argument.

Zdaj pa še vnuki.

```
def vnuki(drevo, ime):
    s = []
    for otrok in otroci(drevo, ime):
        s += otroci(drevo, otrok)
    return s
```

V seznam `s` zbiramo vnuke. Za vsakega otroka v seznam dodamo vse njegove otroke. Če kdo nima otrok, funkcija `otroci` vrne prazen seznam in vse je, kot mora biti.

Kako bi namesto otrok in vnukov iskali starše in stare starše? Natanko enako, le slovar obrnemo v drugo smer. To najpreprosteje storimo tako, da že ob branju datoteke sestavljamo še obratni slovar,

```
drevo_r[otrok].append(stars)
```

Tega potem uporabimo v funkcijah `starsi` in `stari_starsi`, ki bi bili sicer enaki funkcijam otroci in vnuki.

91. Naključno generirano besedilo

Naslednikov ne bomo iskali na enega, temveč na tri načine. Na vse tri pa bomo uporabili slovar s privzetimi vrednostmi, `collections.defaultdict`.

Prvi: z indeksi.

```
import collections
def nasledniki(s):
    nas = collections.defaultdict(list)
    besede = s.split()
    for i in range(len(besede)):
        nas[besede[i - 1]].append(besede[i])
    return nas
```

Besedilo razbijemo na besede, gremo z zanko prek njih in v slovarju za besedo z indeksom `i-1` povemo, da ji sledi (tudi) beseda z indeksom `i`. Ko je `i` enak `0`, je `i-1` enak `-1`; v začetku torej zatrdimo, da zadnji besedi sledi prva. To je povsem praktično; naj vsaki besedi sledi vsaj ena beseda. S tem zagotovimo, da nekaj sledi tudi zadnji, čeprav se v vsem besedilu morda pojavi edinole na zadnjem mestu.

Drugi način: zapomnimo si prejšnjo besedo.

```
def nasledniki(s):
    nas = collections.defaultdict(list)
    besede = s.split()
    prejsnja = besede[-1]
    for beseda in besede:
        nas[prejsnja].append(beseda)
        prejsnja = beseda
    return nas
```

V zanki ponavljamo tole: `prejsnji` besedi sledi trenutna `beseda`; trenutno besedo si, za naslednji krog zanke, zapomnimo kot `prejsnja`. V začetku, pred zanko, rečemo, da je `prejsnja` beseda zadnja beseda iz besedila, `besede[-1]`.

Rešitev je morda videti nerodna, saj potrebujemo dodatno spremenljivko in nekaj več prekladanja. Vendar ima tudi svoje prednosti: predstavljajte si, da besede ne bi bile v seznamu, temveč bi jih, recimo, dobivali po komunikacijskem kanalu (recimo, da opazujemo, kaj nekdo

tipka v Skype). V tem primeru bi postopali natanko tako kot zgoraj: zapomnili bi si prejšnjo besedo in jo povezali z naslednjo. Če je, recimo, besedilo dolgo, si morda ne želimo v resnici sestaviti seznama vseh besed, temveč bomo raje obdržali celoten niz in iz njega jemali besedo za besedo. Objavimo tudi to rešitev, a brez komentarja.

```
import re, collections
def nasledniki(s):
    nas = collections.defaultdict(list)
    prejsnja = ""
    for beseda in re.finditer("\S+", s):
        nas[prejsnja].append(beseda.group())
        prejsnja = beseda.group()
    return nas
```

Že pri več rešitvah smo omenjali "običajne fraze". Oglejte si rešitev naloge Nepadajoči seznami, kjer smo prav tako predlagali dva načina. Če pozabimo na kontekst, vidimo, da gre za eno in isto: tu zlagamo v slovar pare zaporednih besed, tam smo preverjali urejenost zaporednih elementov seznama. Obakrat pa smo imeli na voljo dostopanje do elementov z indeksi ali pa pomnjenje prejšnje besede. Kot smo že povedali: programer le ponavlja ene in iste fraze.

Tretji način: zadrga.

Kdor razume zadrgo, `zip`, naj razume, ostali pač ne bodo.

```
def nasledniki(s):
    nas = collections.defaultdict(list)
    besede = s.split()
    for prejsnja, naslednja in zip(besede, besede[1:]):
        nas[prejsnja].append(naslednja)
    return nas
```

Zdaj pa generiranje besedila. Uporabljali bomo funkcijo `choice` iz modula `random`: damo ji seznam in funkcija naključno izbere enega od elementov.

```
def filozofiraj(besede, dolzina):
    vse_besede = list(besede)
    beseda = random.choice(vse_besede)
    for i in range(dolzina):
        print(beseda, end=" ")
        beseda = random.choice(besede[beseda])
```

V začetku ji pomolimo seznam vseh besed, da izbere prvo. Ker imamo besede v slovarju, jih moramo pretvoriti v seznam; če pokličemo `list(besede)`, dobimo ravno seznam ključev – točno to, kar potrebujemo. Nato ponavljamo tole: izpišemo besedo in kot naslednjo besedo naključno izberemo eno od naslednic trenutne besede.

Funkcija predpostavlja, da ima vsaka beseda vsaj eno naslednico. Če je nima, je najboljše, da filozofiranje nadaljujemo z naključno besedo. To dosežemo tako, da zadnjo vrstico spremenimo v

```
        beseda = random.choice(besede.get(beseda, vse_besede))
```

Če slovar besede ne vsebuje trenutne besede, bo `besede.get` vrnil, kar smo mu dali kot privzeto vrednost, torej celoten seznam besed, in `random.choice` bo izbiral med njimi.

Ploščina, ki se jih zato si lahko poljubno število oseb. Zdaj pa bomo torej je naloga zahteva, da ga pretvorimo v Pythonu pa leto brez nje.

```
def po_starosti(s):
    return len(s)<2 or s[i]%2==1 and not razpored2
```

prvi števki rojstni dnevi zberemo vse tiste, ki so večja ali obratno, ali je popolno število `100 (405//100=4)`. Kdor je enako `0`. Nato gremo čez črke

```
c = 0
while len(osebe) > najvecji:
    najvecji, naj_mesto = []
    for c1, c2 in range(1, 1000)?
```

Take reči BUM. Kadar bo bral knjige ne `8`. Torej bomo izračunali "novi" predzadnji člen, v njem. Če je v običajen slovar.

Da, take reči res BUM. ☺

(Zdaj mi je jasno: veliko študentov ne svojih prispevkov na forume ne sporočil profesorjem v resnici sploh ne piše ročno. Slog gornjega besedila prepoznam že od daleč. To je to. Nezgrešljivo.)

92. Grde besede

Besedilo razdelimo na besede. Če je beseda na seznamu grdih besed, jo zamenjamo z naključno sopomenko, sicer jo pustimo takšno, kot je. (Kar smo pravkar napisali, je pravzaprav ponovitev besedila naloge, istočasno pa tudi po slovensko povedan spodnji program.)

```
import random
def lektor(besedilo, zamenjave):
    novo = ""
    for beseda in besedilo.split():
        if beseda in zamenjave:
            novo += random.choice(zamenjave[beseda]) + " "
        else:
            novo += beseda + " "
    return novo[:-1]
```

Presledke dodajamo za vsako besedo, v `return` pa mimogrede odbijemo odvečni zadnji presledek.

Zdaj pa pride piljenje. Nepotrpežljivi programer bi najbrž naredil takole:

```
def lektor(besedilo, zamenjave):
    novo = ""
    for beseda in besedilo.split():
        novo += random.choice(zamenjave.get(beseda, [beseda])) + " "
    return novo[:-1]
```

Po novem torej zamenjamo vsako besedo. Metoda `get` bo vrnila seznam sopomenk, kadar jih ni, pa seznam, ki vsebuje kar to besedo (in `random.choice` je bo prisiljen izbrati).

Ker tole nudi izhodišče za funkcijo, ki uporabi izpeljan seznam in mimogrede reši problem presledkov, jo le napišimo.

```
def lektor(besedilo, zamenjave):
    return " ".join(random.choice(zamenjave.get(beseda, [beseda])) for beseda in
besedilo.split()))
```

Zdaj pa še funkcija, ki zna ravnati z besedilom, ki poleg besed in presledkov vsebuje tudi ločila. Nekoliko zahtevnejša je.

```
import re, random
def lektor(besedilo, zamenjave):
    return re.sub("[a-z]+",
      lambda x: random.choice(zamenjave.get(x.group(), [x.group()])),
      besedilo)
```

Za tiste, ki želijo razumeti: funkcija `re.sub` prejme tri argumente: prvi je regularni izraz in zadnji besedilo, po katerem iščemo. Drugi argument je lahko niz, s katerim zamenjamo najdeni podniz, ali pa funkcija, ki kot argument prejme objekt, ki ima, med drugim, metodo `group`, s katero lahko dobimo ujemajoče se besedilo (pa še kaj drugega). Tule smo `sub` poklicali z na hitro definirano funkcijo, ki zamenja najdeno besedo z naključno izbrano lepo sopomenko.

93. Kronogrami

Program obupanca:

```
def kronogram(napis):
    letnica = 0
    for znak in napis:
        if znak == "I":
            letnica += 1
        elif znak == "V":
            letnica += 5
        elif znak == "X":
            letnica += 10
        elif znak == "L":
            letnica += 50
        elif znak == "C":
            letnica += 100
        elif znak == "D":
            letnica += 500
        elif znak == "M":
            letnica += 1000
    return letnica
```

V mnogih drugih jezikih bi tule uporabili stavek `switch`. Python ga nima, zato ga, očitno, ne moremo uporabiti. Vendar ga redko pogrešamo, saj (skoraj) vedno obstaja lepa rešitev brez njega. Tule, recimo, imamo boljšo rešitev s seznamom znakov in njihovih vrednosti.

Za vsako črko, ki kaj šteje, preštejemo, kolikokrat se pojavi v nizu in k vsoti prištejemo tolikokratno vrednost znaka. Če se, recimo, pojavijo trije znaki "L", prištejemo 3*50.

```
stevke = [("I", 1), ("V", 5), ("X", 10), ("L", 50),
          ("C", 100), ("D", 500), ("M", 1000)]

def kronogram(napis):
    letnica = 0
    for znak, vrednost in stevke:
        letnica += vrednost * napis.count(znak)
    return letnica
```

Lahko pa obrnemo: namesto, da za vsak znak, ki kaj šteje, pogledamo, kolikokrat se pojavi, naredimo zanko prek znakov niza in za vsak znak, ki je kaj vreden, prištejemo, kolikor je vreden.

```
stevke = {"I": 1, "V": 5, "X": 10, "L": 50,
          "C": 100, "D": 500, "M": 1000}

def kronogram(napis):
    letnica = 0
    for znak in napis:
        if znak in stevke:
            letnica += stevke[znak]
    return letnica
```

Drobna variacija na to temo se znebi pogojnega stavka. Spomnimo se: slovarjevi metodi `get` lahko podamo privzeto vrednost, ki jo vrne, če iskanega ključa ni v slovarju. Znaki, ki jih nismo našteli, imajo vrednost 0.

```
def kronogram(napis):
    letnica = 0
    for znak in napis:
        letnica += stevke.get(znak, 0)
    return letnica
```

Resen programer bi izračune po obeh variantah opravil v eni vrstici. S seznamom,

```
stevke = [("I", 1), ("V", 5), ("X", 10), ("L", 50),
          ("C", 100), ("D", 500), ("M", 1000)]

def kronogram(napis):
  return sum(vrednost * napis.count(znak) for znak, vrednost in stevke)
```

in s slovarjem,

```
stevke = {"I": 1, "V": 5, "X": 10, "L": 50,
          "C": 100, "D": 500, "M": 1000}
def kronogram(napis):
    return sum(stevke.get(znak, 0) for znak in napis)
```

Naloga ima tudi nekaj zabavnih rešitev. V nizu lahko zamenjamo vse znake V s petimi `I`-ji, vse znake X z desetimi... in tako naprej in na koncu vse znake M s 1000 I-ji. Potem preštejemo, koliko I-jev imamo.

```
def kronogram(napis):
    return napis.replace("V", "I" * 5).replace("X", "I" * 10). \
        replace("L", "I" * 50).replace("C", "I" * 100).
        replace("D", "I" * 500).replace("M", "I" * 1000).count("I")
```

V naslednji funkciji nikjer ne piše, da je I vreden 1, V 5, X 10... pa vendar deluje. Kako to?

```
def kronogram(s):
    v = 0
    for c in s:
        i = "IVXLCDM".find(c)
        v += i>=0 and (1 + 4 * (i % 2)) * 10 ** (i // 2)
    return v
```

Ta čudna rešitev nas pripelje do še čudnejše. Temu, ki jo razume, bo gotovo še posebej všeč prvi argument funkcije `map`.

```
def kronogram(s):
    return sum(i>=0 and (1 + 4 * (i % 2)) * 10 ** (i // 2)
               for i in map("IVXLCDM".find, s))
```

94. Posebnež

Ta naloga je lepa, ker ima toliko različnih rešitev.

Pripravimo lahko slovar `koliko_kdo`, katerega ključi so različna števila bombonov, vrednosti pa seznam otrok, ki ima to število bombonov. Če imajo, recimo, Ana, Berta in Cilka po tri bonbone, Dani pa štiri, bomo dobili slovar `{3: ["Ana", "Berta", "Cilka"], 4: ["Dani"]}`.

```
import collections
def posebnez(s):
    koliko_kdo = collections.defaultdict(list)
    for otrok, bombonov in s.items():
        koliko_kdo[bombonov].append(otrok)
    for kdo in koliko_kdo.values():
        if len(kdo) == 1:
            return kdo[0]
```

V drugi zanki se sprehodimo čez vrednosti v tem slovarju - ključi nas niti ne zanimajo, potrebovali smo jih le, ko smo slovar sestavljali. Pravzaprav vemo, kako bodo videti: ena vrednost bo seznam vseh otrok razen enega, druga vrednost pa seznam s tem otrokom. Poiščemo torej seznam z enim samim elementom in vrnemo prvi (in seveda edini) element tega slovarja.

Zanimiva povsem drugačna rešitev je tale: sestavimo seznam parov (stevilo_bombonov, ime_otroka) in ga uredimo. Posebnež bo zdaj prvi ali zadnji - glede na to, ali ima več ali manj bombonov kot ostali. Če imata prva dva otroka enako bombonov, vrnemo ime zadnjega, sicer ime prvega.

```
def posebnez(s):
    po_bombonih = [(bombon, otrok) for otrok, bombon in s.items()]
    po_bombonih.sort()
    if po_bombonih[0][0] == po_bombonih[1][0]:
        return po_bombonih[-1][1]
    else:
        return po_bombonih[0][1]
```

Program je malo zoprn zaradi igre indeksov. Podobnih rešitev je še veliko. Lahko vzamemo, recimo, tri otroke in preverimo, koliko bombonov imajo. Če je eden različen, vrnemo njegovo ime. Če jih imajo vsi trije enako, gremo čez slovar in poiščemo otroka, ki jih nima toliko...

Tule ste še dve lepi.

Vrednosti slovarja (`s.values()` zložimo v množico (`set(s.values())`). Imela bo dva elementa. To množico spremenimo v seznam (`list(set(s.values()))`) in iz nje poberemo tadva elementa v `a` in `b`, takole: `a, b = list(set(s.values()))`. Tisti, ki ima posebno število bombonov, ima bodisi `a` bodisi `b` bombonov. Preverimo, ali je takšnih, ki imajo `b` bombonov, samo 1: vsa števila bombonov pretvorimo v seznam (`list(s.values())`) in preštejemo, koliko `b`-jev vsebuje. Če le enega, potem prepišemo `b` v `a`.

Po tem je v `a` število bombonov, ki jih vsebuje posebnež. Odtod naprej je preprosto.

```
def posebnez(s):
    a, b = list(set(s.values()))
    if list(s.values()).count(b) == 1:
        a = b
    for ime, bombonov in s.items():
        if bombonov == a:
            return ime
```

Tale je pa najlepša: vzamemo prve tri elemente seznama in jih uredimo po velikosti. Srednji element gotovo vsebuje toliko bombonov, kolikor jih vsi razen posebneža. Zakaj? Če med prvimi tremi elementi ni posebneža, potem ima srednji element prav gotovo "neposebno" število bombonov. Če je posebnež med prvimi tremi, pa je na začetku (ker ima manj bombonov kot ostali) ali pa na koncu (ker jih ima več). Prav gotovo pa ni na sredi.

Ko vemo, koliko bombonov imajo vsi, razen posebneža, gremo čez slovar in vrnemo tistega, ki ima drugačno število bombonov.

```
def posebnez(s):
    navadno = sorted(s.values())[1]
    for ime, bombonov in s.items():
        if bombonov != navadno:
            return ime
```

95. Sumljive besede

Načrt je takšen: najprej razbijemo stavek na besede. Potem pripravimo slovar, v katerem bomo za vsak znak beležili, v koliko besedah se je pojavil (da se ne hecamo z detajli, bomo

uporabili že dobro znani `defaultdict`, še boljšemu `Counter` se bomo izognili, da ne bomo preveč specifični za Python); gremo prek besed in za vsako črko, ki se pojavi v besedi, povečamo števec. Nato gremo prek števcev, da poiščemo črko, ki se je pojavila v vseh besedah razen v eni. Končno gremo ponovno prek besed in vrnemo tisto, ki te črke nima.

```
from collections import defaultdict
def sumljiva(stavek):
    besede = stavek.split()
    v_besedah = defaultdict(int)
    for beseda in besede:
        for crka in set(beseda):
            v_besedah[crka] += 1
    for crka, pogostosti in v_besedah.items():
        if pogostosti == len(besede) - 1:
            break
    for beseda in besede:
        if not crka in beseda:
            return beseda
```

Za prvi del poskrbi metoda split. Če bi hoteli rešiti težjo različico, bi bilo najpreprosteje uporabiti regularne izraze (več nalog z njimi bomo videli na koncu, a tudi takrat naj vas ne vznemirja, če jih ne boste razumeli); napisali bi `besede = re.findall("\w+")`.

Drugi del je skoraj običajno preštevanje, a s trikom. Gremo prek besed. Črke vsake besede spravimo v množico, da se bo vsaka črka pojavila le enkrat (v besedi "beseda" imamo dva e-ja, v množici bo le eden).

Odtod naprej je trivialno: za vsak par crka, pogostost, ki ga priskrbi `v_besedah.items()` preverimo, ali je pogostost enaka `len(besede)-1`: to drži natančno pri črki, ki se pojavi v vseh besedah razen v eni. Tedaj prekinemo zanko (`break`) in spremenljivka `crka` bo vsebovala črko, ki jo iščemo. V zadnji zanki gremo prek besed; ko naletimo na tisto brez črke `crka`, jo vrnemo; z `return` je seveda konec tudi zanke in z njo funkcije.

96. Transakcije

Naloga preskuša, ali znamo narediti in uporabiti slovar. Kot pravi namig, najprej pretvorimo vse skupaj v slovar; to lahko naredimo z zanko `for` ali, še preprosteje, s klicem funkcije `dict`.

Nato gremo z zanko prek transakcij, odštejemo denar prvemu in ga preštejemo drugemu. Končno v tem slovarju poiščemo tistega z največ denarja. Z malo spretnosti bi lahko uporabili kar funkcijo `max`, v spodnji rešitvi pa smo se odločili kar za običajno zanko.

```
def transakcije(zacetek, trans):
    stanje = dict(zacetek)
    for kdo, komu, koliko in trans:
        stanje[kdo] -= koliko
        stanje[komu] += koliko
    najvec = None # lahko bi napisali -1, a morda so vsi v dolgovih...
    for kdo, koliko in stanje.items():
        if najvec is None or koliko > najvec:
            najvec, kdo_naj = koliko, kdo
    return kdo_naj
```

97. Natakar

Naredimo slovar, se sprehodimo z zanko čez naročila in dodajamo ali preklicujemo jedi.

```
import collections

def narocila(s):
    po_strankah = collections.defaultdict(list)
    for oseba, jed in s:
        if jed[0] == "-":
            if jed[1:] in po_strankah[oseba]:
                po_strankah[oseba].remove(jed[1:])
        else:
            po_strankah[oseba].append(jed)
    return dict(po_strankah)
```

Brez slovarjev s privzetimi vrednostmi bi morali takoj znotraj zanke preveriti, ali osebo že poznamo. Če je ne, bi jo dodali.

```
        if oseba not in po_strankah:
            po_strankah[oseba] = []
```

98. Tečaji

Čim razumemo, kaj naj bi bilo v slovarju (in če smo že kdaj uporabljali slovarje s privzetimi vrednostmi), je opravil trivialna: le tečaj doda v ustrezno množico.

```
def opravil(ime, tecaj, tecaji):
    tecaji[ime].add(tecaj)
```

`najbolj_ucen` ponavlja ničkolikokrat izvedeni dril: iskanje največjega elementa. Če znamo uporabiti argument `key` v funkciji `max`, lahko napišemo kar

```
def najbolj_ucen(tecaji):
    def deg(k):
        return len(tecaji[k])
    return max(tecaji, key=deg)
```

ali celo

```
def najbolj_ucen(tecaji):
    return max(tecaji, key=lambda k: len(tecaji[k]))
```

Če ne, pa se potrudimo z

```
def najbolj_ucen(tecaji):
    naj = None
    for kdo, komu in tecaji.items():
        if naj is None or len(komu) > len(tecaji[naj]):
            naj = kdo
    return naj
```

Funkcija `vsi_tecaji` le zloži vse vrednosti v isto množico.

```
def vsi_tecaji(tecaji):
    vsi = set()
    for dst in tecaji.values():
        vsi |= dst
    return vsi
```

Neopravljeni tečaji pa so vsi razen opravljenih.

```
def neopravljeni(ime, tecaji):
    return vsi_tecaji(tecaji) - tecaji[ime]
```

Resen programer bi te funkcije napisal takole:

```
def opravil(ime, tecaj, tecaji):
    tecaji[ime].add(tecaj)

def najbolj_ucen(tecaji):
    return max(tecaji, key=lambda k: len(tecaji[k]))

from functools import reduce
def vsi_tecaji(tecaji):
    return reduce(set.union, tecaji.values(), set())

def neopravljeni(ime, tecaji):
    return vsi_tecaji(tecaji) - tecaji[ime]
```

99. Lego

Sprehoditi se moramo čez slovar potrebnih reči. Da dobimo pare (stvar, količina) bomo uporabili metodo `items()` (šlo bi tudi brez nje, a z njo je lepše). Za vsako stvar pogledamo, ali je v škatli in ali je je dovolj. Če je ni ali če je ni dovolj, vrnemo `False`. Sicer pa pustimo zanko, da se izteče do konca in če se, na koncu vrnemo `True`.

```
def vsi_deli(skatla, potrebno):
    for stvar, kolicina in potrebno.items():
        if stvar not in skatla or skatla[stvar] < potrebno[stvar]:
            return False
    return True
```

Slovar manjkajočih stvari naredimo na zelo podoben način: gremo čez vse potrebno. Če je nekaj v škatli, vendar v premajhni količini, zabeležimo, koliko te stvari še potrebujemo (toliko, kolikor je je potrebno, minus toliko, kolikor je dobimo v škatli. Če pa nečesa ni v škatli, potrebujemo še vse.

```
def kaj_manjka(skatla, potrebno):
    manjka = {}
    for stvar, kolicina in potrebno.items():
        if stvar in skatla:
            if skatla[stvar] < kolicina:
                manjka[stvar] = kolicina - skatla[stvar]
        else:
            manjka[stvar] = kolicina
    return manjka
```

Gre pa tudi krajše - če vemo, kaj počne metoda `get`: vrne vrednost, shranjeno pod določenim ključem, če je ni v slovarju, pa vrne privzeto vrednost, ki jo podamo kot drugi element metodi `get`.

```
def kaj_manjka(skatla, potrebno):
    manjka = {}
    for stvar, kolicina in potrebno.items():
        potrebujemo = kolicina - skatla.get(stvar, 0)
        if potrebujemo > 0:
            manjka[stvar] = potrebujemo
    return manjka
```

Nalogo je mogoče rešiti tudi tako, da najprej napišemo drugo funkcijo in jo kličemo iz prve.

```
def vsi_deli(skatla, potrebno):
    return not kaj_manjka(skatla, potrebno)
```

100. Slepa polja

Gremo čez vsa polja. Za vsako polje pogledamo polja, v katera je mogoče priti. Če ta polja ne nastopajo v slovarju, so slepa in jih dodamo v množico slepih polj.

```
def slepa_polja(s):
    slepa = set()
    for kam in s.values():
        for t in kam:
            if t not in s:
                slepa.add(t)
    return slepa
```

101. Inventar

Funkcija, ki vrne zalogo, mora le prebrati vrednost iz inventarja.

```
def zaloga(inventar, izdelek):
    return inventar[izdelek]
```

Ne le, da se zgodi, celo dogaja se, da ljudje pišejo neumnosti, kot je tale:

```
def zaloga(inventar, izdelek):
    for art, kol in inventar.items():
        if art == izdelek:
            return kol
```

Slovarje imamo prav zato, da nam takšnih stvari ni potrebno pisati. Bistvo slovarjev je prav v tem, da znajo hitro poiskati vrednost, ki pripada določenemu ključu.

Funkcija, ki zmanjša zalogo, mora zmanjšati količino podanega izdelka v inventarju.

```
def prodaj(inventar, izdelek, kolicina):
    inventar[izdelek] -= kolicina
```

Bistvo te naloge je, da spreminjamo slovar, ne pa vračamo novi slovar.

Ob računanju primanjkljaja, samo pomislimo, kako bi to delal pravi skladiščnik. Vzel bi prazen list, na katerega si bo zapisoval, kaj manjka (`manjka = {}`). Za vsako stvar iz seznama naročil (`for izdelek in narocilo`) bo pogledal, če ga ima na zalogi (`if izdelek in inventar`). V tem primeru pogleda, ali ga ima slučajno premalo (`if narocilo[izdelek] > inventar[izdelek]`). Če je tako, zabeleži primankljaj (`manjka[izdelek] = narocilo[izdelek] - inventar[izdelek]`). Če ga je na zalogi dovolj, ne stori nič. Sicer, torej, če izdelka sploh ni na zalogi, je primanjkljaj takšen, kolikor tega izdelka je naročenega (`manjka[izdelek] = narocilo[izdelek]`). Strnjeno,

```
def primanjkljaj(inventar, narocilo):
    manjka = {}
    for izdelek in narocilo:
        if izdelek in inventar:
            if narocilo[izdelek] > inventar[izdelek]:
                manjka[izdelek] = narocilo[izdelek] - inventar[izdelek]
        else:
            manjka[izdelek] = narocilo[izdelek]
    return manjka
```

Pazite, kam smo postavili `else`: pod tisti `if`, na katerega se nanaša!

Ta rešitev je čisto spodobna. Če poznamo malo več Pythona, pa naredimo tako:

```
def primanjkljaj(inventar, narocilo):
    manjka = {}
    for izdelek, kolicina in narocilo.items():
        imamo = inventar.get(izdelek, 0)
        if imamo < kolicina:
            manjka[izdelek] = kolicina - imamo
    return manjka
```

Klic `inventar.get(izdelek, 0)` vrne vrednost, ki pripada ključu `izdelek`, če izdelka ni v slovarju, pa vrne `0`. Tako torej dobimo njegovo zalogo (ali 0, če ga nimamo) in se znebimo kupa pogojev.

102. Nedostopna polja

Najprej rečemo, da so vsa polja nedostopna: shranimo jih v množico. To storimo z `nedostopna = set(s)` – če naredimo `set(s)`, dobimo ravno množico `s`-jevih ključev.

Nato od te množice odštevamo vsa polja, ki so dostopna - zapisana so kot vrednosti v slovarju. Ker so tudi to množice, jih lahko preprosto odštevamo od množice nedostopnih polj.

```
def nedostopna(s):
    ned = set(s)
    for v in s.values():
        ned -= v
    return ned
```

103. Opravljivke

Najprej `kam(cevi, cev)`. Dokler podana cev kam vodi (kar vemo po tem, da se pojavi med ključi v slovarju, kar preverimo s `cev in cevi`), gremo v to cev (`cev = cevi[cev]`). Ko je konec, vrnemo cev, kjer smo se ustavili (`return cev`).

```
def kam(cevi, cev):
    while cev in cevi:
        cev = cevi[cev]
    return cev
```

Sestavimo števec, recimo z `defaultdict`, ki bo prešteval, koliko kroglic je dobila posamezna izhodna cev. Nato gremo prek vhodnih cevi, pogledamo, kam vodijo (za to lahko uporabimo gornjo funkcijo) in povečamo ustrezni števec za 1.

```
def koliko_kam(cevi, zacetne):
    koliko = defaultdict(int)
    for cev in zacetne:
        koliko[kam(cevi, cev)] += 1
    return koliko
```

Če ne poznamo defaultdict-a – običajne komplikacije. Poleg tega opozorimo, da tole ni preveč dobra ideja:

```
def koliko_kam(cevi, zacetne):
    koliko = defaultdict(int)
    for cev in zacetne:
        if not kam(cevi, cev) in koliko:
            koliko[kam(cevi, cev)] = 1
        else:
            koliko[kam(cevi, cev)] += 1
    return koliko
```

Kolikokrat ta funkcija namreč pokliče funkcijo `kam`? Če ima seznam `zacetne` *n* elementov, bo funkcija *2n*-krat poklicala `kam`. Le rezultat prvega klica si moramo zapomniti, pa bo dvakrat hitrejša.

```
def koliko_kam(cevi, zacetne):
    koliko = defaultdict(int)
    for cev in zacetne:
        kam_gre = kam(cevi, cev)
        if not kam_gre in koliko:
            koliko[kam_gre] = 1
        else:
            koliko[kam_gre] += 1
    return koliko
```

Ko imamo funkcijo `koliko_kam`, je pisanje funkcije `najpogostejsa` prava rutina. Gremo prek slovarja, ki ga vrne `koliko_kam` ter si zapomnimo največjo vrednost in pripadajoči ključ. Tule je ena od različic.

```
def najpogostejsa(cevi, zacetne):
    naj = None
    kolikam = koliko_kam(cevi, zacetne)
    for katera, koliko in kolikam.items():
        if naj is None or koliko > kolikam[naj]:
            naj = katera
    return naj
```

V začetku postavimo `naj` na `None`, kar nam bo povedalo, da nismo pogledali še nobene cevi. Potem izračunamo, koliko jih gre kam (pri čemer seveda uporabimo prejšnjo funkcijo). Nato gremo čez vse pare končnih cevi in njihovih števcev. V pogoju poglejmo najprej drugi del,

`koliko > kolikam[naj]`: če je v "to" končno cev (`katera`) prišlo več krogel kot v tisto, v katero jih je prišlo največ, si zapomnim, da jih je prišlo največ v to (`naj = katera`). To bi bilo lepo in prav, samo za prvo cev ne deluje. Na kaj naj postavim `naj` v začetku? Ena možnost je torej narediti, kot smo naredili: v začetku ga postavimo na `None` in potem še preden preverimo ali je

`koliko > kolikam[naj]`, preverimo, ali je `naj` slučajno enak `None`. Če je, potem si seveda zapomnimo to cev kot najpopularnejšo, saj je pač prva doslej.

Mimogrede, nismo napisali `naj == None`, temveč `naj is None`. V tem primeru v bistvu ni razlike, po nekih pravilih lepega vedenja v Pythonu pa je zaželeno, da takrat, kadar imamo samo en objekt določene vrste (takšni so, recimo, `None`, `False` in `True` – vsi `True`-ji, recimo, niso samo *enaki* `True`-ji temveč celo *isti* `True`), uporabljamo `is` in ne `==`. Če boste sami uporabljali `==`, ni nič narobe, knjiga pa mora dajati dober zgled.

104. Viri in ponori

Tale je pa trivialna, če malo pomislimo:

- začetne cevi so tiste, ki se pojavljajo med ključi, ne pa tudi med vrednostmi,
- končne cevi so tiste, ki se pojavljajo med vrednostmi in ne med ključi.

Takšne reči se lepo pove z množicami. Na srečo (in ne po naključju) tudi naloga zahteva množice teh cevi, ne seznamov.

```
def zacetne(cevi):
    return set(cevi.keys()) - set(cevi.values())

def koncne(cevi):
    return set(cevi.values()) - set(cevi.keys())
```

Metode keys() pravzaprav ne potrebujemo: če iz slovarja naredimo množico, bomo tako ali tako dobili množico ključev.

```
def zacetne(cevi):
    return set(cevi) - set(cevi.values())

def koncne(cevi):
    return set(cevi.values()) - set(cevi)
```

Mimogrede, če bi naloga slučajno zahtevala sezname, to sicer ne bi bil poseben problem – le rezultat bi spremenili v seznam.

```
def zacetne(cevi):
    return list(set(cevi) - set(cevi.values()))

def koncne(cevi):
    return list(set(cevi.values()) - set(cevi))
```

105. Ponori

Reševanje te naloge je mogoče zelo (in brez potrebe) zaplesti. Začnemo lahko s končnimi cevmi (vrne jih funkcija `koncne`) in potem za vsako od njih gledamo, katera cev vodi do njih.

Veliko lažje je v nasprotno smer. Gremo prek vseh cevi, končnih in začetnih in vseh vmes – vsega, kar se pojavlja v slovarju. To dobimo s `set(cevi.keys()) | set(cevi.values())` in ne slučajno, `zacetne(cevi) | koncne(cevi)`, saj to izpusti cevi, ki niso ne začetne in ne končne.

Za vsako preverimo, kam vodi (s funkcijo `kam`) in zabeležimo, da v množico, ki pripada tej končni cevi, dodamo to začetno.

```
def viri(cevi):
    v = defaultdict(set)
    for zacetek in cevi.keys() | cevi.values():
        v[kam(cevi, zacetek)].add(zacetek)
    return v
```

Hm, tule piše `cevi.keys() | cevi.values()` in ne `set(cevi.keys()) | set(cevi.values())`. Operator `|` je operator za unijo množic, vendar `cevi` in `cevi.values()` vendar nista množici. Kako potem to deluje? Da ne bi kdo mislil, da Python sam od sebe pretvarja tipe (tega ne počne, saj vendar ni Javascript!), le povejmo, v katero smer pogledati (seveda je to namenjeno le radovednežem, ki jih gledanje v takšne smeri zanima): `cevi.keys()` in `cevi.values()` sta generatorja. Operator `|`, če ga uporabimo na generatorji, postavi generatorja enega za drugega. Ko se prvi "iztroši", se vključi drugi.

106. Sociogram

Za vse pare oseb in njihovih prijateljev (`for oseba, prijatelji in mreza.items()`) je bilo potrebno pogledati, ali je ciljna oseba (`komu`) med `prijatelji` (`if komu in prijatelji`) in v tem primeru dodati med prijatelje te osebe še `osebo` (`s.add(oseba)`).

```
def prijatelji(komu, mreza):
    s = set()
    for oseba, prijatelji in mreza.items():
        if komu in prijatelji:
            s.add(oseba)
    return s
```

Nadaljevanje je nekaj, kar gledam skoz in skoz: funkcija, ki išče največji element in vrne nekaj, kar pripada temu elementu. Rešitev je čisto klasična.

```
def najbolj_priljubljen(mreza):
    kdo = None
    koliko = 0
    for k in mreza:
        kk = len(prijatelji(k, mreza))
        if kk > koliko:
            koliko = kk
            kdo = k
    return kdo
```

107. Izplačilo

Bankovce se najprej splača urediti po velikosti, od največjega proti najmanjšemu. To storimo s `sorted(bankovci.items(), reverse=True)`; konkretno, tole bo dalo urejen seznam parov (vrednost, število bankovcev).

Zdaj gremo lepo po bankovcih v tem vrstnem redu: `for bankovec, kolicina in sorted(bankovci.items(), reverse=True)`. Bankovec nas zanima le, če je znesek, ki ga želimo izplačati z njim, večji od bankovca (`if znesek >= bankovec`). (Ta pogoj ni čisto potreben, a ga obdržimo zaradi preglednosti. Delovalo pa bi tudi brez njega.)

Izračunamo, koliko kosov tega bankovca bi bilo potrebno izplačati. Uporabimo celoštevilsko deljenje. Če bi izplačali toliko, kolikor jih imamo ali pa celo več, si zapomnimo, da jih bomo izplačali, kolikor jih imamo in pobrišemo bankovec iz slovarja; sicer pa število bankovcev v slovarju zmanjšamo za toliko, kolikor jih bomo izplačali. Na koncu zmanjšamo še znesek za toliko, kolikor je vreden bankovec * število kosov, ki smo jih izplačali.

```
def izplacilo(bankovci, znesek):
    for bankovec, kolicina in sorted(bankovci.items(), reverse=True):
        if znesek >= bankovec:
            koliko = znesek // bankovec
            if koliko >= bankovci[bankovec]:
                koliko = bankovci[bankovec]
                del bankovci[bankovec]
            else:
                bankovci[bankovec] -= koliko
            znesek -= bankovec * koliko
```

Vse skupaj mirno ponavljamo prek vseh bankovcev. Če znesek slučajno že predtem pade na 0, nas to ne vznemirja: bomo pač pustili vse ostale bankovce pri miru.

108. Dopisovalci

Čim razumemo, kaj naj bi bilo v slovarju (in če smo že kdaj uporabljali slovarje s privzetimi vrednostmi), je funkcija `dopis` trivialna: le dopisovalca doda.

```
def dopis(src, dst, relacije):
    relacije[src].add(dst)
```

Funkcija `najzgovornejsi` ponavlja ničkolikokrat izvedeni dril: iskanje največjega elementa. Pa jo zdaj napišimo malo napredneje: uporabimo funkcijo max, ki jih kot dodatni argument key podamo ključ, po katerem naj primerja elemente. Ključ pa bo predstavljala funkcija deg, ki vrne dolžino vrednosti, ki pripada podanemu ključu.

```
def najzgovornejsi(relacije):
    def deg(k):
        return len(relacije[k])
    return max(relacije, key=deg)
```

Kdor ne zna tako, pa je še vedno lahko naredil tako, kot v sto in eni drugi funkciji, recimo tako:

```
def najzgovornejsi(relacije):
    naj = None
    for kdo, komu in relacije.items():
        if naj is None or len(komu) > len(relacije[naj]):
            naj = kdo
    return naj
```

Funkcija `vse_osebe` le zloži v isto množico vse ključe in vse vrednosti. Ključe lahko dobi kar s `set(relacije)`, za množice se mora sprehoditi prek vrednosti.

```
def vse_osebe(relacije):
    vsi = set(relacije)
    for dst in relacije.values():
        vsi |= dst
    return vsi
```

Neznanci pa so vsi, razen tistih, ki jim je dotični pisal in dotičnega samega.

```
def neznanci(src, relacije):
    return vse_osebe(relacije) - relacije[src] - {src}
```

V praksi bi resen programer v Pythonu te funkcije napisal takole:

```
def dopis(src, dst, relacije):
    relacije[src].add(dst)

def najzgovornejsi(relacije):
    return max(relacije, key=lambda k: len(relacije[k]))

from functools import reduce
def vse_osebe(relacije):
    return reduce(set.union, relacije.values(), set(relacije))

def neznanci(src, relacije):
    return vse_osebe(relacije) - relacije[src] - {src}
```

109. Zaporniki

Kako ugotovimo, ali dva soseda govorita kak skupni jezik, je namignila naloga: presek množic jezikov mora biti neprazen. Ostalo so le zanke - prva po parih sosednjih stolpcev, druga po parih sosednjih vrstic.

```
def sogovorniki(s):
    ista = 0
    for v in s:  # za vsako vrstico ...
        for i, j in zip(v, v[1:]):  # ...poglej sosednji celici
            if set(i) & set(j):
                ista += 1
    for v1, v2 in zip(s, s[1:]):  # Za vsak par vrstic ...
        for i, j in zip(v1, v2):  # ... poglej istoležna stolpca
            if set(i) & set(j):
                ista += 1
    return ista
```

Zdaj pa še malo "poenostavimo", če se temu lahko tako reče. Če vemo, da je `True` isto kot `1` in `False` isto kot `0`; če vemo, da se logičnim vrednostim reče `bool`; če vemo, da objekte tipa `str`, `int`, `float`, `list`, ... dobimo tako, da pokličemo funkcije (tipe, konstruktorje - imenujte jih kakor jih hočete) `str`, `int`, `float`, `list`, ..., se rešitev še nekoliko poenostavi.

```
def sogovorniki(s):
    ista = 0
    for v in s:
        for i, j in zip(v, v[1:]):
            ista += bool(set(i) & set(j))
    for v1, v2 in zip(s, s[1:]):
        for i, j in zip(v1, v2):
            ista += bool(set(i) & set(j))
    return ista
```

To vodi v rešitev, ki le prešteje, kar je treba.

```
def sogovorniki(s):
    return sum(bool(set(i) & set(j))
               for v in s for i, j in zip(v, v[1:]))
               +
               sum(bool(set(i) & set(j)) for v1, v2 in zip(s, s[1:])
               for i, j in zip(v1, v2))
```

Ali pa sestavimo vse komunicirajoče pare in jih preštejemo.

```
def sogovorniki(s):
    return len([(i, j) for v in s for i, j in zip(v, v[1:]) if set(i) & set(j)] +
               [(i, j) for v1, v2 in zip(s, s[1:])
                for i, j in zip(v1, v2) if set(i) & set(j)])
```

110. Ograje

Najprej razčistimo, koliko bi bilo ograj, če bi imela vsa polja različne lastnike. Če je polje dimenzije m×n, potrebujemo `(n+1)*m` navpičnih ograj in `(m+1)*n` vodoravnih, torej skupaj `(n+1)*m + (m+1)*n`. Lahko pa obrnemo drugače in vsakemu polju postavimo levo in zgornjo ograjo; teh je `2*m*n`, poleg tega pa spodnjim poljem še spodnjo in desnim še desno, zato `2*m*n + m + n`. Oboje je isto.

Sledita zanki, ki pregledujeta sosednje celice po vrsticah in po stolpcih. Vse je zelo podobno prejšnji nalogi, zato le napišimo rešitev.

```
def ograje(s):
    ista = 0
    for v in s:
        for i, j in zip(v, v[1:]):
            if i == j:
                ista += 1
    for v1, v2 in zip(s, s[1:]):
        for i, j in zip(v1, v2):
            if i == j:
                ista += 1
    return 2 * len(s) * len(s[0]) + len(s) + len(s[0]) - ista
```

Če vemo, da je `True` isto kot `1` in `False` isto kot `0`, se rešitev še nekoliko poenostavi.

```
def ograje(s):
    ista = 0
    for v in s:
        for i, j in zip(v, v[1:]):
            ista += i == j
    for v1, v2 in zip(s, s[1:]):
        for i, j in zip(v1, v2):
            ista += i == j
    return 2 * len(s) * len(s[0]) + len(s) + len(s[0]) - ista
```

To vodi v rešitev, ki le prešteje, kar je treba.

```
def ograje(s):
    return 2 * len(s) * len(s[0]) + len(s) + len(s[0]) - \
           sum(i == j for v in s for i, j in zip(v, v[1:])) - \
           sum(i == j for v1, v2 in zip(s, s[1:]) for i, j in zip(v1, v2))
```

111. Trgovanje

Ali obstaja kdo, ki menja to za ono, lahko preverimo tako, da se zapeljemo čez ponudbe in takrat, ko najdemo takšno, ki ustreza, vrnemo `True`. Če ni nobene, vrnemo `False`.

```
def obstaja(dam, dobim, ponudbe):
    for dobi, da in ponudbe:
        if dam in dobi and dobim in da:
            return True
    return False
```

Menjave gredo ravno obratno: čim najdem takšno, ki ni možna, vrnem `False`. Če takšne ni, vrnem `True`.

```
def menjave(zaporedje, ponudbe):
    for dam, dobim in zip(zaporedje, zaporedje[1:]):
        if not obstaja(dam, dobim, ponudbe):
            return False
    return True
```

Tu gre za dva vzorca, ki ju gledamo že od začetka, recimo od takrat, ko smo iskali praštevila. Če znamo uporabljati generatorje, za ta vzorca uporabimo `any` in `all`.

```
def obstaja(dam, dobim, ponudbe):
    return any(dam in dobi and dobim in da
               for dobi, da in ponudbe)

def menjave(zaporedje, ponudbe):
    return all(obstaja(dam, dobim, ponudbe)
               for dam, dobim in zip(zaporedje, zaporedje[1:]))
```

112. Ne na lihih

Iščemo množico vseh elementov (`set(s)`), razen tistih, ki so na lihih mestih (`s[1::2]`), torej

```
def ne_na_lihih(s):
   return set(s) - set(s[1::2])
```

113. Sopomenke

Strategija je preprosta: `split`, zamenjave, `join`.

Če iščemo sopomenke za neko besedo, bi bilo dobro imeti slovar sopomenk za vse besede. Dobimo ga tako

```
sopomenke1 = {}
for i, so in enumerate(sopomenke):
    for beseda in so:
        sopomenke1[beseda] = list(so - {beseda})
```

Sestavili bomo torej slovar `sopomenke1`, katerega ključi so besede, pripadajoče vrednosti pa seznami sopomenk te besede (brez te besede same). Gremo torej čez seznam terk s sopomenkami. Za vsako množico besed gremo čez te besede in v slovar dodamo seznam besed iz te množice - brez te besede.

Zdaj, ko smo to obdelali, spišimo celo funkcijo:

```
import random

def predelaj(stavek, sopomenke):
    sopomenke1 = {}
    for so in sopomenke:
        for beseda in so:
            sopomenke1[beseda] = list(so - {beseda})

    novi_stavek = []
    for b in stavek.split():
        if b in sopomenke1:
            novi_stavek.append(random.choice(sopomenke1[b]))
        else:
            novi_stavek.append(b)
    return " ".join(novi_stavek
```

V drugem delu razkopljemo stavek na besede. Za vsako preverimo, ali je v slovarju sopomenk in v tem primeru v `novi_stavek` (ki je v resnici seznam besed, ki bodo v novem stavku), dodamo naključno izbrano sopomenko. Če beseda nima sopomenk, dodamo kar to besedo.

Na koncu te besede zlepimo v nov stavek.

Večino drugega dela,

```
        if b in sopomenke1:
            novi_stavek.append(random.choice(sopomenke1[b]))
        else:
            novi_stavek.append(b)
```

lahko skrajšamo, če se spomnimo na `get`:

```
        novi_stavek.append(random.choice(sopomenke1.get(b, [b])))
```

Tale `sopomenke1.get(b, [b])` bo v primeru, da slovar nima sopomenk, vrnil seznam `[b]`, torej seznam, ki vsebuje originalno besedo in `random.choice` ne bo imel veliko izbire, saj bo vrnil kar originalno besedo. Tako dobimo

```
import random
def predelaj(stavek, sopomenke):
    sopomenke1 = {}
    for so in sopomenke:
        for beseda in so:
            sopomenke1[beseda] = list(so - {beseda})
    novi_stavek = []
    for b in stavek.split():
        novi_stavek.append(random.choice(sopomenke1.get(b, [b])))
    return " ".join(novi_stavek)
```

Če se nočemo zafrkavati s po nepotrebnem predolgimi funkcijami, pa napišemo kar

```
import random
def predelaj(stavek, sopomenke):
    sopomenke1 = {beseda: list(so - {beseda})
                  for so in sopomenke for beseda in so}
    return " ".join(random.choice(sopomenke1.get(b, [b]))
                    for b in stavek.split())
```

114. Stavka z istim pomenom

Rešitev je malo zavozlana.

Najprej preverimo, ali imata stavka slučajno različno število besed in vrnemo `False`, če to ne drži. Sicer gremo čez vse pare. Če sta besedi različni, gremo čez vse sopomenke. Če najdemo množico sopomenk, v kateri se pojavita obe besedi, zanko prekinemo. Če se zanka izteče, ne da bi jo prekinili, pa gre za različni besedi, ki nista sopomenki, torej vrnemo `False`.

Če se zanka `for beseda1, beseda2` prekine z `break` izteče, se `return False` ne bo izvedel, zato vrnemo `True`.

```
def sopomena(stavek1, stavek2, sopomenke):
    stavek1, stavek2 = stavek1.split(), stavek2.split()
    if len(stavek1) != len(stavek2):
        return False
    for beseda1, beseda2 in zip(stavek1, stavek2):
        if beseda1 != beseda2:
            for so in sopomenke:
                if beseda1 in so and beseda2 in so:
                    break
            else:
                return False
    return True
```

Lepo bi bilo, če bi se izognili stalnemu sprehajanju po slovarju sopomenk. Lahko bi, recimo, sestavili slovar sopomenskih parov besed.

Po bližnjici se to naredi tako:

```
pari = {(b1, b2) for so in sopomenke for b1 in so for b2 in so}
```

V tem slovarju so zdaj, recimo `{("pob", "fant"), ("fant", "pob"), ("pob", "pob"), ("fant", "fant"), ("dekle", "punca), ...}`.

Zdaj pa naredimo pare besed iz obeh stavkov: `zip(stavek1.split(), stavek2.split())`. Pravzaprav jih moramo dati v množico: `set(zip(stavek1.split(), stavek2.split()))`. Vsak par besed mora biti tudi v slovarju parov sopomenk. Z drugimi besedami, množica teh parov mora biti podmnožica množice sopomenk. Tako dobimo

```
def sopomena(stavek1, stavek2, sopomenke):
    return set(zip(stavek1.split(), stavek2.split())) <=
               {(b1, b2) for so in sopomenke for b1 in so for b2 in so}
```

Če hočemo paziti na dolžine, dodamo še

```
def sopomena(stavek1, stavek2, sopomenke):
    stavek1, stavek2 = stavek1.split(), stavek2.split()
    return len(stavek1) == len(stavek2) and \
        set(zip(stavek1, stavek2)) <=
            {(b1, b2) for so in sopomenke for b1 in so for b2 in so}
```

115. Združi – razmeči

S prvim delom naloge bomo opravili brez težav.

```
from collections import defaultdict

def zdruzi(s):
    res = defaultdict(set)
    for i, e in enumerate(s):
        res[e].add(i)
    return res
```

Če se ne spomnimo `defaultdict` potrebujemo še malo čaranja z `if`.

Poučen primer napačne rešitve je:

```
def zdruzi(s):
    rdic = {}
    for number in s:
        if number in rdic:
            rdic[number].append(s.index(number))
        else:
            rdic[number] = [s.index(number)]
    return rdic
```

Metoda `index` je nevarna. Prvič, če lahko vemo, kje v seznamu smo – in kadar gremo čez seznam, vedno lahko vemo, kje smo, saj imamo `enumerate` – ne uporabljamo `index`. Opozoril sem, da `index` ne bo deloval, če imamo v seznamu več enakih elementov, saj bo vračal prvega. Točno to se zgodi v tej funkciji. Še več, tu je očitno, da se bo ista vrednost pojavila večkrat, saj vsa naloga govori prav o tem. `s.index(number)` bo vedno vračal indeks prve pojavitve.

Razmetavanje pa je druga pesem. V osnovi bi želeli napisati tole:

```
def razmeci(s):
    res = []
    for e, places in s.items():
        for place in places:
            res[place] = e  # Ne deluje!
    return res
```

Tole ne deluje, ker `res[place]` še ne obstaja, saj je seznam v začetku prazen. Pač pa bi delovalo, če bi vnaprej poznali dolžino novega seznama, saj bi potem lahko napisali

```
def razmeci(s):
    res = [None] * dolzina_novega_seznama  # Ne deluje!
    for e, places in s.items():
        for place in places:
            res[place] = e
    return res
```

No, lahko jo izračunamo. Lahko gremo, recimo, čez vse indekse in pogledamo največjega

```
def razmeci(s):
    naj = 0
    for indeksi in s.values():
        m = max(indeksi)
        if m > naj:
            naj = m
    res = [None] * (naj + 1)
    for e, places in s.items():
        for place in places:
            res[place] = e
    return res
```

To je nekoliko dolgo, pišemo lahko

```
def razmeci(s):
    res = [None] * (1 + max(max(indeksi) for indeksi in s.values()))
    for e, places in s.items():
        for place in places:
            res[place] = e
    return res
```

Nič od tega ne deluje, če je seznam prazen. To lahko rešimo z dodanim `if`-om na začetku, lahko pa namesto največjega indeksa preštejemo število indeksov in prazen seznam se uredi sam od sebe.

```
def razmeci(s):
    res = [None] * sum(map(len, s.values()))
    for e, places in s.items():
        for place in places:
            res[place] = e
    return res
```

Pri tej nalogi se lahko domislimo veliko ustvarjalnih rešitev. Ena lepših – napisal jo je nek študent – je tale:

```
def razmeci(s):
    vmesni = []
    for number, index in s.items():
        for i in index:
            vmesni.append((i, number))

    rezultat = []
    for i, j in sorted(vmesni):
        rezultat.append(j)
    return rezultat
```

V seznam `vmesni` zložimo pare indeksov in vrednost. Nato ga uredimo (po indeksih) in v rezultat prepišemo vrednosti v prav tem vrstnem redu.

Rešitev v tej obliki kar kliče po tem, da jo zapišemo z izpeljanim seznamom.

```
def razmeci(s):
    return [v for i, v in sorted((i, v)
            for v, indeksi in s.items()
            for i in indeksi)]
```

116. Podajanje daril

Za reševanje te naloge ne potrebujemo slovarjev. Kdo poda komu, izvemo z zanko

```
for kdo, komu in pari:
    if kdo == x:
        x = komu
        break
```

Zanki dodamo še `else`, ki naj se izvede, če `kdo` ne podaja nikomur. Tako dobimo rešitev:

```
def dolzina_poti(pari, x):
    podaj = 0
    while True:
        for kdo, komu in pari:
            if kdo == x:
                x = komu
                break
        else:
            return podaj
        podaj += 1
```

Okrog zanke zgoraj smo dodali neskončno zanko `while` s števcem podaj. Prekinila se bo z `else` po `for`-u, torej, ko kdo ne podaja naprej. Takrat funkcija vrne število podaj.

Da, ta rešitev ni zahtevala slovarja. Vendar ne deluje, če je seznam res dolg. Ko so študenti reševali to nalogo na izpitu, so v testnih primerih dobili seznam dolžine 50000. In ta rešitev se ni obnesla. Težava je v počasnosti zanke, ki preverja, kdo podaja komu. Če se je znebimo, bo funkcija postala veliko hitrejša.

Kdo podaja komu moramo prepisati v slovar. Lahko tako

```
kdo_komu = {}
for kdo, komu in pari:
    kdo_komu[kdo] = komu
```

Lahko pa se spomnimo, kako sestavljamo slovarje z `dict` in napišemo kar

```
kdo_komu = dict(pari)
```

Funkcija bo zdaj povsem preprosta.

```
def dolzina_poti(pari, x):
    kdo_komu = dict(pari)
    podaj = 0
    while x in kdo_komu:
        podaj += 1
        x = kdo_komu[x]
    return podaj
```

117. Požrešneži

Tokrat bomo uporabili množice: sestavimo množico vseh, ki dobivajo čokolade in množico vseh, ki jih dajejo. Naloga hoče, da vrnemo razliko.

```
def ne_daje_naprej(pari):
    dobi = set()
    daje = set()
    for kdo, komu in pari:
        dobi.add(komu)
        daje.add(kdo)
    return dobi - daje
```

Če se hočemo znebiti zanke, vržemo – po zgledu prejšnje rešitve – pare v slovar. Nato od množice vrednosti odštejemo množico ključev.

```
def ne_daje_naprej(pari):
    pari = dict(pari)
    return set(pari.values()) - set(pari)
```

118. Ne maram

Najpreprostejša rešitev je, da pač za vsak par zaporednih elementov preverimo, ali je na seznamu prepovedanih parov ali ne. Popaziti moramo, da se lahko zgodi, da prva noče biti z drugo ali pa druga s prvo.

```
def preveri_vrsto(vrsta, prepovedani):
    for a, b in zip(vrsta, vrsta[1:]):
        if (a, b) in prepovedani or (b, a) in prepovedani:
            return False
    return True
```

Ker je `prepovedani` seznam, bo iskanje po njem počasno. Funkcijo bistveno pospešimo, če ga spremenimo v množico.

```
def preveri_vrsto(vrsta, prepovedani):
    prepovedani = set(prepovedani)
    for a, b in zip(vrsta, vrsta[1:]):
        if (a, b) in prepovedani or (b, a) in prepovedani:
            return False
    return True
```

Če pa smo pri množicah: čemu ne bi spremenili še parov, ki jih imamo v vrsti, v množico. Tedaj nas zanima le, ali je presek teh dveh množic prazen: če je, bodo uvrščenke zadovoljne.

```
def preveri_vrsto(vrsta, prepovedani):
        return not (set(zip(vrsta, vrsta[1:])) | set(zip(vrsta[1:], vrsta))) \
                   & set(prepovedani)
```

119. Najmanjši unikat

Preštejemo, koliko je katerih znakov. Zato lahko, kot navadno, uporabimo `defaultdict`, ali pa, še lažje, `Counter`. Vzamemo vse pare (element, število ponovitev) in jih uredimo. Urejeni bodo po velikosti prvih elementov terke (`element`). Nato vrnemo prvi element, ki se ponovi le enkrat.

```
def najmanjsi_unikat(s):
    stevci = Counter(s)
    for e, c in sorted(stevci.items()):
        if c == 1:
            return e
```

Študente, ki so reševali to nalogo, je zelo skrbelo, da bodo takrat, ko ni nobenega ne-unikatnega elementa, z `return None` vrnili `None`. Ni treba. Tudi gornja funkcija vrne `None` – pač tako, da ne vrne ničesar.

Če znamo uporabljati generatorje, pa bo program bistveno krajši.

```
def najmanjsi_unikat(s):
    return min((e for e, c in Counter(s).items() if c == 1), default=None)
```

120. Bingo

Rešitev naloge je veliko. Lahko, recimo, gledamo množico doslej izžrebanih številk. Po vsakem žrebanju preverimo, ali so številke na kakem izmed listkov podmnožica doslej izžrebanih. Če je tako, vrnemo ta listek.

```
def bingo(listki, vrstni_red):
    izklicano = set()
    for stevilka in vrstni_red:
        izklicano.add(stevilka)
        for listek in listki:
            if set(listek) <= izklicano:
                return listek
```

121. Trki besed

Na prvo žogo nalogo rešimo tako, da pač preverimo vse pare besed. Za to potrebujemo dvojno zanko, znotraj katere preverimo, ali se besedi skrijeta v isto besedo, pri čemer mora seveda iti za različni besedi.

```
def trki(xs):
    for beseda1 in xs:
        for beseda2 in xs:
            if beseda1 != beseda2 and skrij(beseda1) == skrij(beseda2):
                return beseda1, beseda2
```

Težava te funkcije je, da bo za 10000 besed 200,000.000-krat (10000×10000×2) poklicala funkcijo `skrij`.

Tudi če jo izboljšamo tako, da drugo besedo išče le med besedami, ki so na seznamu pred prvo, tako da bo vsak par pogledala le enkrat, in tudi če prvo besedo skrijemo le enkrat,

```
def trki(xs):
    for i, beseda1 in enumerate(xs):
        for beseda2 in xs[:i]:
            if skrij(beseda1) == skrij(beseda2):
                return beseda1, beseda2
```

nismo pridobili veliko, saj je program le dvakrat hitrejši – namesto dvesto milijonkrat bo poklicala `skrij` "samo" sto milijonkrat (točneje, 99,990.000-krat). Še enkrat hitrejši bomo, če ne kličemo `skrij(beseda1)` znotraj notranje zanke, temveč znotraj zunanje.

```
def trki(xs):
    for i, beseda1 in enumerate(xs):
        skrita1 = skrij(beseda1)
        for beseda2 in xs[:i]:
            if skrita1 == skrij(beseda2):
                return beseda1, beseda2
```

Zdaj `skrij` pokličemo petdeset milijonkrat (točneje, 10000 + 10000×9999/2 = 50.005,000).

Vse to računanje služi temu, da povemo, kako boljša je rešitev s slovarjem. Vsako skrito besedo, ki jo naračunamo, vtaknemo v slovar - skrita beseda je ključ, vrednost pa beseda, iz katere jo dobimo. Vsakič, ko pridelamo novo skrito besedo, preverimo, ali smo jo že videli (ključi) in vrnemo besedo, iz katere je nastala (vrednost, ki pripada temu ključu).

```
def trki(xs):
    d = {}
    for x in xs:
        k = skrij(x)
        if k in d:
            return d[k], x
        d[k] = x
```

Ta funkcija je desettisočkrat hitrejša od prve, saj pokliče funkcijo `skrij` le tolikokrat, kolikorkrat je treba. Cena, ki jo moramo plačati za to, je, da porabi precej več pomnilnika.

Naloga ima zanimivo ozadje: če bi znali dobro iskati takšne pare pri nekaterih točno določenih vrstah funkcij, bi znali ponarejati elektronske podpise, certifikate spletnih mest...

Rekurzija

122. Preštej vnuke

Trik te naloge je: ni rekurzije, čeprav je v razdelku rekurzija. ;)

Iti nam je prek vse otrok in seštevati število njihovih otrok.

```
def prestej_vnuke(oseba):
    otroci = rodovnik[oseba]
    vnukov = 0
    for otrok in otroci:
        vnukov += len(rodovnik[otrok])
    return vnukov
```

Če oseba nima otrok, se zanka ne izvede nikoli in vse je OK. Če otrok nima otrok, je dolžina seznama njegovih otrok 0 in spet je vse v redu.

Hitreje se tej reči streže takole:

```
def prestej_vnuke(oseba):
    return sum(len(rodovnik[otrok]) for otrok in rodovnik[oseba])
```

123. Poišči rojaka

Če imamo sreči in je osebi, katere rojaka iščemo, ime `ime`, je to to – vrnemo `True`. Sicer vsakega od otrok vprašamo, ali je v njegovi rodbini kdo z imenom `ime`. Tam se zgodba ponovi: otrok bo imel srečo ali pa vprašal svoje otroke.

```
def poisci_rojaka(oseba, ime):
    if oseba == ime:
        return True
    otroci = rodovnik[oseba]
    for otrok in otroci:
        if poisci_rojaka(otrok, ime):
            return True
    return False
```

Spet gre tudi krajše, veliko krajše.

```
def poisci_rojaka(oseba, ime):
    return oseba == ime or \
           any(poisci_rojaka(otrok, ime) for otrok in rodovnik[oseba])
```

124. Poišči potomca

Pozor, past! Nekdo bi naivno rekel: saj to je tako, kot `poisci_rojaka`, le da ne preverjamo osebe, temveč le njegove potomce.

```
# Napačna rešitev!
def poisci_potomca(oseba, ime):
    otroci = rodovnik[oseba]
    for otrok in otroci:
        if poisci_potomca(otrok, ime):
            return True
    return False
```

To ne deluje: vsaka oseba le vpraša svoje otroke, ali je med njimi kateri s podanim imenom. Otroci vprašajo otroke – pa čeprav imajo morda sami pravo ime ... in tako naprej do tistih, ki nimajo otrok. Ta funkcija vrne `True` le, kadar dobi `True` iz rekurzivnega klica, od "tam spodaj" pa vedno dobi le `False`.

Očitni rešitvi ste dve. Prva: vsaka oseba vrne `True`, če je iskano ime med njegovimi (neposrednimi) otroki. Če ga ni, vpraša, ali je med njihovimi.

```
def poisci_potomca(oseba, ime):
    otroci = rodovnik[oseba]
    if ime in otroci:
        return True
    for otrok in otroci:
        if poisci_potomca(otrok, ime):
            return True
    return False
```

Za drugo je potrebno imeti funkcijo `poisci_rojaka`. Ime se pojavi med potomci osebe, če se pojavi v rodbinah njegovih sinov.

```
def poisci_potomca(oseba, ime):
    otroci = rodovnik[oseba]
    for otrok in otroci:
        if poisci_rojaka(otrok, ime):
            return True
    return False
```

Funkcija `poisci_potomca` zdaj sploh ni rekurzivna, pač pa kliče rekurzivno funkcijo `poisci_rojaka`.

Spet gre tudi krajše.

```
def poisci_potomca(oseba, ime):
    return any(poisci_rojaka(otrok, ime) for otrok in rodovnik[oseba])
```

125. Preštej rodbino

Vsaka oseba šteje sebe (`rodbina = 1`) in k temu prišteje velikosti rodbin svojih otrok.

```
def prestej_rodbino(oseba):
    rodbina = 1
    otroci = rodovnik[oseba]
    for otrok in otroci:
        rodbina += prestej_rodbino(otrok)
    return rodbina
```

Krajše pa tako:

```
def prestej_rodbino(oseba):
    return 1 + sum(prestej_rodbino(otrok) for otrok in rodovnik[oseba])
```

126. Preštej potomce

Tudi tu nas čaka enaka past kot pri nalogi Poišči potomca. Naivno bi lahko vzeli funkcijo `prestej_rodbino`, a spremenili `rodbina = 1` v `rodbina = 0`, tako da ne štejemo trenutne osebe, temveč le potomce.

```
# Napačna rešitev!
def prestej_potomce(oseba):
    potomcev = 0
    otroci = rodovnik[oseba]
    for otrok in otroci:
        potomcev += prestej_potomce(otrok)
    return potomcev
```

Tole je še očitneje narobe: funkcija računa neke vsote nečesa ... a to so vsote samih ničel. Nikjer nobene enice.

Naivna rešitev je takšna, da sicer šteje tudi osebo, vendar na koncu odšteje 1.

```
def prestej_potomce(oseba):
    potomcev = 1
    otroci = rodovnik[oseba]
    for otrok in otroci:
        potomcev += prestej_potomce(otrok)
    return potomcev - 1
```

S tem nismo pridobili prav ničesar. Vse, kar smo pridobili, ko namesto `potomcev = 0` pišemo `potomcev = 1`, uničimo s tem, ko namesto `return potomcev` pišemo `return potomcev - 1`.

Rešitev iz pasti je enaka kot prej. Vsaka oseba ima, najprej, toliko potomcev, kolikor ima otrok, k temu pa je potrebno prišteti še potomce vsakega otroka.

```
def prestej_potomce(oseba):
    otroci = rodovnik[oseba]
    potomcev = len(otroci)
    for otrok in otroci:
        potomcev += prestej_potomce(otrok)
    return potomcev
```

Ali, krajše,

```
def prestej_potomce(oseba):
    otroci = rodovnik[oseba]
    return len(otroci) + sum(prestej_potomce(otrok) for otrok in otroci)
```

Včasih takšna rešitev ne bo vžgala. Včasih je kakšno funkcijo preprosto težko napisati v rekurzivni obliki, pač pa lahko napišemo sorodno rekurzivno funkcijo. Tule sicer nimamo opravka s takšnim primerom, a si vendarle oglejmo še rešitev v tem slogu. Podobno kot smo pri iskanju imena med potomci lahko uporabili iskanje ime v rodbini, lahko tudi tu za preštevanje potomcev uporabimo preštevanje rodbine: število potomcev je za 1 manjše od velikosti rodbine.

```
def prestej_potomce(oseba):
    return prestej_rodbino(oseba) - 1
```

127. Najdaljše ime v rodbini

Rešitev je precej podobna iskanju velikosti rodbine, le da namesto vsote iščemo maksimum. Najprej vzamemo ime osebe; to je najdaljše ime doslej. Ker lahko ime vsebuje tudi različne številke in krajevne nadimke, vzamemo le prvi del imena, `oseba.split()[0]`. Nato vsakega otroka vprašamo po najdaljšem imenu v njegovi rodbini (vključno z njegovim). Če je daljše od najdaljšega doslej, si ga zapomnimo.

```
def najdaljse_ime(oseba):
    otroci = rodovnik[oseba]
    najdaljsi = oseba.split()[0]
    for otrok in otroci:
        najdaljsi_1 = najdaljse_ime(otrok)
        if len(najdaljsi_1) > len(najdaljsi):
            najdaljsi = najdaljsi_1
    return najdaljsi
```

Tule moramo popaziti, da funkcijo `najdaljse_ime` pokličemo le enkrat. Če bi pisali

```
def najdaljse_ime(oseba):
    otroci = rodovnik[oseba]
    najdaljsi = oseba.split()[0]
    for otrok in otroci:
        if len(najdaljse_ime(otrok)) > len(najdaljsi):
            najdaljsi = najdaljse_ime(otrok)
    return najdaljsi
```

bi dobili grozljivo počasno funkcijo: ne bi bila le dvakrat počasnejša, čas izvajanja bi naraščal kar eksponentno. Ko znotraj `najdaljse_ime` dvakrat (namesto enkrat) pokličemo `najdaljse_ime`, se tudi znotraj teh klicev funkcija pokliče po dvakrat, torej skupaj štirikrat. Tudi znotraj klicev znotraj klicev se funkcija pokliče po dvakrat, torej osemkrat namesto enkrat... Na globini 5, recimo, bomo funkcijo poklicali 32-krat namesto enkrat.

128. Globina rodbine

Vsakega otroka vprašamo po globini njegove rodbine. Če je večja od največje doslej, si jo zapomnimo. Na koncu vrnemo globino, povečano za 1, tako da vključi še osebo `oseba`.

```
def globina(oseba):
    otroci = rodovnik[oseba]
    najglobji = 0
    for otrok in otroci:
        najglobji_1 = globina(otrok)
        if najglobji_1 > najglobji:
            najglobji = najglobji_1
    return 1 + najglobji
```

Spet velja enako svarilo kot v prejšnji nalogi: funkcijo `globina` smemo znotraj funkcije `globina` poklicati le enkrat.

Z generatorji gre krajše.

```
def globina(oseba):
    otroci = rodovnik[oseba]
    if otroci:
        return 1 + max(globina(otrok) for otrok in otroci)
    else:
        return 1
```

Ali pa kar

```
def globina(oseba):
    otroci = rodovnik[oseba]
    return len(otroci) and 1 + max(globina(otrok) for otrok in otroci)
```

129. Kolikokrat ime

Ime se, najprej, pojavi enkrat, če je osebi `oseba` ime `ime` in ničkrat, če ji ni. K temu je potrebno prišteti število pojavitev tega imena v rodbinah vseh otrok.

```
def kolikokrat_ime(oseba, ime):
    otroci = rodovnik[oseba]
    if oseba.split()[0] == ime
        kolikokrat = 1
    else:
        kolikokrat = 0
    for otrok in otroci:
        kolikokrat += kolikokrat_ime(otrok, ime)
    return kolikokrat
```

Če se spomnimo, da sta True in False isto kot 0 in 1, lahko reč še malo skrajšamo.

```
def kolikokrat_ime(oseba, ime):
    otroci = rodovnik[oseba]
    kolikokrat = int(oseba.split()[0] == ime)
    for otrok in otroci:
        kolikokrat += kolikokrat_ime(otrok, ime)
    return kolikokrat
```

Funkcija bi delovala tudi, če bi izpustili klic `int`. Moti nas le, da bi takrat, kadar oseba nima otrok in se zanka ne izvrši nikoli, namesto `0` ali `1` vračala `True` ali `False`.

Če znamo uporabljati generatorje, pa še malo bolj.

```
def kolikokrat_ime(oseba, ime):
    return oseba.split()[0] == ime + \
           sum(kolikokrat_ime(otrok, ime) for otrok in rodovnik[oseba])
```

Klic `int` zdaj ni več potreben, saj se `bool` spremeni v `int` ob prištevanju.

130. Koliko žensk

Preštevanje žensk v rodbini je nesramno enaka reč kot preštevanje rodbine, le da ne štejemo tistih članov, katerih ime se ne konča z "a".

```
def zensk_v_rodbini(oseba):
    otroci = rodovnik[oseba]
    if oseba.split()[0][-1] == "a":
        zensk = 1
    else:
        zensk = 0
    for otrok in otroci:
        zensk += zensk_v_rodbini(otrok)
    return zensk
```

Tudi tu lahko uberemo enake bližnjice kot v prejšnji nalogi. Ne bomo jih ponavljali.

131. Naštej rodbino

Rodbino določene osebe dobimo tako, da k tej osebi prištejemo rodbine njenih otrok. Vse je podobno štetju potomcev, le da funkcija ne vrača števil, temveč množice. Vzamemo torej `prestej_rodbino`, spremenimo `rodbina = 1` v `rodbina = {oseba}` in `+=` spremenimo v `|=`, da namesto vsote izračunamo unijo.

```
def nastej_rodbino(oseba):
    rodbina = {oseba}
    otroci = rodovnik[oseba]
    for otrok in otroci:
        rodbina |= nastej_rodbino(otrok)
    return rodbina
```

132. Naštej potomce

Tudi naštevanje potomcev je podobno preštevanju potomcev, le namesto dolžine seznama otrok, `len(otroci)`, nas zanima množica otrok, `set(otroci)`.

```
def vse_potomstvo(oseba):
    otroci = rodovnik[oseba]
    potomstvo = set(otroci)
    for otrok in otroci:
        potomstvo |= vse_potomstvo(otrok)
    return potomstvo
```

Ako se je kdo zmotil in napisal `{otroci}` je namesto množice otrok dobil množico, ki je vsebovala seznam otrok. Če imamo, recimo, `otroci = [Ana, Berta]`, je `{otroci}` pač `{["Ana", "Berta"]}`. Da dobimo, kar hočemo, je potrebno klicati "funkcijo" `set`; tej damo seznam nekih reči (ali slovar nekih reči ali niz nekih črk ali kaj podobnega) in to spremeni v množico teh reči.

133. Največ otrok

Tole je spet skoraj kot preštevanje potomcev, le da števila ne seštevamo, temveč iščemo njihov maksimum. Najprej predpostavimo, da je največ otrok toliko, kolikor jih ima oseba (`najvec = len(otroci)`). Nato gremo prek otrok, poizvemo, kolikšna je največja družina v rodbini vsakega od njih. Če je večja od trenutno največje, si jo zapomnimo. Na koncu vrnemo, kar smo pač največ dobili.

```
def najvec_otrok(oseba):
    otroci = rodovnik[oseba]
    najvec = len(otroci)
    for otrok in otroci:
        st_otrok = najvec_otrok(otrok)
        if st_otrok > najvec:
            najvec = st_otrok
    return najvec
```

134. Največ vnukov

Preštevanje števila vnukov nam ne bi smelo dati preveč vetra. S tem lahko opravimo po normalni poti,

```
def vnukov(oseba):
    vnukov = 0
    for otrok in rodovnik[oseba]:
        vnukov += len(rodovnik[otrok])
    return vnukov
```

ali po bližnjici

```
def vnukov(oseba):
    return sum(len(rodovnik[otrok]) for otrok in rodovnik[oseba])
```

Naprej pa gre natančno tako kot v nalogi, kjer smo iskali največje število otrok, le `len(oseba)` zamenjamo z `vnukov(oseba)`.

```
def najvec_vnukov(ime, rodovnik):
    najvec = vnukov(ime, rodovnik)
    for otrok in rodovnik[ime]:
        otrokovih = najvec_vnukov(otrok, rodovnik)
        if otrokovih > najvec:
            najvec = otrokovih
    return najvec
```

135. Največ sester

Najprej funkcija, ki pove, koliko sester ima najbolj osestreni potomec podane osebe. Prešteti moramo, koliko hčera ima oseba. Če ima toliko hčera, kolikor ima otrok, ima vsaka od njih toliko sester, kolikor je otrok - 1, saj nobena ni sestra sama sebi. Če pa je med otroki tudi kak sin, ima toliko sester, kolikor je žensk. Z drugimi besedami,

```
def sester_pod(ime, rodovnik):
    otroci = rodovnik[ime]
    sester = len([otrok for otrok in otroci if otrok.split()[0][-1] == "a"])
    if sester and sester == len(otroci):
        sester -= 1
    return sester
```

Če vemo, da je True isto kot 1 in False isto kot 0, lahko to nekoliko skrajšamo v

```
def sester_pod(ime, rodovnik):
    otroci = rodovnik[ime]
    sester = len([otrok for otrok in otroci if otrok.split()[0][-1] == "a"])
    return sester – (sester == len(otroci))
```

Odtod gre po znanem vzorcu: najprej pogledamo, koliko sester ima najbolj osestreni otrok podane osebe, nato povprašamo otroke po najbolj osestrenih ljudeh iz njihovih rodbin in vrnemo maksimum vsega skupaj.

```
def najvec_sester(ime, rodovnik):
    najvec = sester_pod(ime, rodovnik)
    for otrok in rodovnik[ime]:
        sester = najvec_sester(otrok, rodovnik)
        if sester > najvec:
            najvec = sester
    return najvec
```

136. Najplodovitejši

Pa smo naleteli na primer, kjer si nam splača pomagati tako, da najprej napišemo malo drugačno funkcijo: napisali bomo funkcijo, ki vrača, prav tako kot prejšnja funkcija, število otrok, poleg tega pa še, kdo je ta srečnež.

```
def najvec_otrok_kdo_koliko(oseba):
    otroci = rodovnik[oseba]
    najvec, naj_kdo = len(otroci), oseba
    for otrok in otroci:
        st_otrok, kdo = najvec_otrok(otrok)
        if st_otrok > najvec:
            najvec, naj_kdo = st_otrok, kdo
    return najvec, naj_kdo
```

Funkcija je v resnici enaka prejšnji, le skupaj s številko ves čas prekladamo tudi ime.

Vendar to ni to, kar zahteva naloga: zahteva samo ime. To se uredi tako, da napišemo še zahtevano funkcijo, ki pa le pokliče tole in vrne drugi element, ime.

```
def najvec_otrok_kdo(oseba):
    return najvec_otrok_kdo_koliko(oseba)[0]
```

137. Brez potomca

Stvar je preprostejša, kot je videti. Brez potomca je lahko oseba sama. Če ni, pa naj vpraša svojega prvega otroka. Ali zadnjega. Ali drugega, če jih ima več. Ali do naključnega – v vsakem primeru bo enkrat prišel do konca, do nekoga brez potomcev.

```
def brez_potomca(oseba):
    if not oseba.otroci:
        return oseba.ime
    else:
        return brez_potomca(oseba.otroci[0])
```

Posebne potrebe po tem, da bi bila funkcija rekurzivna, niti ni. Do prvega prvega prvega prvega potomca brez potomcev bi se lahko sprehodili tudi z zanko `while`.

```
def brez_potomca(oseba):
    while oseba.otroci:
        oseba = oseba.otroci[0]
    return oseba.ime
```

138. Vsi brez potomca

Nalogo rešimo točno po navodilih: če oseba nima otrok, vrne množico s sabo. Če jih ima, vpraša otroke, kateri od njih (oziroma njih potomcev) so brez potomcev, združuje rezultate vsega tega povpraševanja v unijo.

```
def brez_potomcev(oseba):
    if not oseba.otroci:
        return {oseba.ime}
    brez = set()
    for otrok in oseba.otroci:
        brez |= brez_potomcev(otrok)
    return brez
```

139. Kako daleč

Če je `oseba` kar oseba, ki jo iščemo, vrnemo 1. Sicer gremo prek otrok te osebe in vsakega posebej vprašamo, kako globoko je iskano ime. Če vrne kako številko večjo ali enako 0, vrnemo prav to število, le za 1 jo povečamo – če je nekdo za otroka osebe `oseba` na globini 3, je za osebo `oseba` na globini 4. Če vrne -1, pa v rodbini dotičnega otroka ni iskanega imena, torej iščemo naprej. Če se zanka izteče brez uspeha, vrnemo `-1`.

```
def globina_do(oseba, ime):
    if oseba == ime:
        return 0
    otroci = rodovnik[oseba]
    for otrok in otroci:
        globina = globina_do(otrok, ime)
        if globina > -1:
            return globina+1
    return -1
```

140. Pot do

Tole je čisto podobno, le da namesto s številkami opletamo s seznami: funkcija, ki kaj najde, vrne pot. Če nek otrok sporoči pot do osebe, oseba le še doda sebe na začetek seznama, ki ga vrne.

```
def pot_do(oseba, ime):
    if oseba == ime:
        return [oseba]
    otroci = rodovnik[oseba]
    for otrok in otroci:
        pot = pot_do(otrok, ime)
        if pot:
            return [oseba] + pot
```

Če ne najdemo ničesar, ne vrnemo ničesar, torej vrnemo `None`, kakor hoče naloga.

141. Zaporedja soimenjakov

Naloga je težka, ker ... no, saj ni, samo malo bolj jo je potrebno razmisliti, ker ne gre čisto po kalupih prejšnjih rešitev. Preprosta rešitev ima dva dela, nerekurzivnega in rekurzivnega.

```
def najdaljse_zaporedje (oseba):
    naj_z, naj_o = 1, oseba.split()[0]
    otrok = oseba
    while True:
        for otrok in rodovnik[otrok]:
            if otrok.split()[0] == naj_o:
                naj_z += 1
                break
        else:
            break

    for otrok in rodovnik[oseba]:
        naj_z_1, naj_o_1 = najdaljse_zaporedje(otrok)
        if naj_z_1 > naj_z:
            naj_z, naj_o = naj_z_1, naj_o_1
    return naj_z, naj_o
```

V prvi zanki pogledamo, ali ima oseba sina z enakim imenom in ta svojega sina z enakim in tako naprej. Ob predpostavki, da ima vsak Friderik samo enega sina z imenom Friderik, lahko to reč naredimo brez rekurzije. Za to potrebujemo, uh, dve zanki. Zanka `while` gre v globino: v vsakem koraku zanke gremo eno generacijo globje v rodbini. In vsakič naredimo tole: z zanko `for` gremo prek vseh otrok v upanju, da odkrijemo katerega, čigar prvi del imena je enak prvemu delu imena osebe `oseba`, ki ga imamo shranjenega v `naj_o`. Če najdemo takšnega otroka, povečamo `naj_z`, dolžino zaporedja imen. Ime tega otroka je ostalo shranjeno v spremenljivki otrok in v naslednji iteraciji zanke `while` bomo iskali ustrezno poimenovanega potomca med njegovimi otroki (`for otrok in rodovnik[otrok]`).

Če se zanka `for` izteče, ne da bi našli ustrezno poimenovanega otroka, z `else: break` prekinemo zanko `while`.

Ostanek je klasičen: v `naj_z` in `naj_o` imamo dolžino zaporedja ime, ki sledi osebi oseba in njeno ime. Sprehodimo se prek otrok in pogledamo, ali se med njimi najde kaj daljšega. Na koncu vrnemo najdaljše, kar smo odkrili.

Rekli smo, da je naloga nekoliko hujši izziv. Je, vendar ne zaradi rekurzivnega dela – zapleteno je tisto, kar smo reševali z iteracijo.

142. Fakulteta

Tu pa ni kaj komentirati, ta rekurzija je čisto matematična.

```
def fakulteta(n):
    if n == 0:
        return 1
    return n * fakulteta(n - 1)
```

Z uporabo if-else gre še malo krajše.

```
def fakulteta(n):
    return n * fakulteta(n - 1) if n > 0 else 1
```

143. Fibonacijeva števila

Tudi to je trivialno.

```
def fibonaci(n):
    if n < 2:
        return 1
    else:
        return fibonaci(n - 1) + fibonaci(n - 2)
```

ali, krajše,

```
def fibonaci(n):
    return fibonaci(n - 1) + fibonaci(n - 2) if n > 2 else 1
```

144. Vsota

Nalogo rešimo natančno po navodilih: "*Vsota elementov praznega seznama je 0. Vsota nepraznega seznama je enaka vsoti prvega elementa in vsoti preostalega seznama (`s[1:]`).*"

```
def vsota(stevila):
    if not stevila:
        return 0
    return stevila[0] + vsota(stevila[1:])
```

Mimogrede smo se spomnili še, da so prazni seznami neresnični. Če se ne bi, bi pač pisali `if stevila == []`.

Rešitev lahko spet skrajšamo v

```
def vsota(stevila):
    return stevila[0] + vsota(stevila[1:]) if stevila else 0
```

ali kar v

```
def vsota(stevila):
    return len(stevila) and stevila[0] + vsota(stevila[1:])
```

145. Iskanje elementa

Spet gremo kar po navodilih: "*Seznam vsebuje element `x`, če `s` ni prazen in je bodisi prvi element `s` enak `x` bodisi ostanek seznama vsebuje `x`.*"

```
def vsebuje(s, x):
    return s and s[0] == x or vsebuje(s[1:], x)
```

146. Palindrom

Funkcija bo vrnila, kar zahteva naloga: "*niz je palindrom, če je krajši od dveh znakov ali pa je prvi znak enak zadnjemu in je niz med drugim in predzadnjim znakom palindrom*".

```
def palindrom(s):
    return len(s) < 2 or s[0] == s[-1] and palindrom(s[1:-1])
```

147. Enaka seznama

Naloga pravi, da sta seznama `s` in `t` enaka, če sta oba prazna (`not s and not t`) ali pa oba neprazna (`s and t`) in imata enaka prva elementa (`s[0] == t[0]`) in sta enaka tudi ostanka (`enaka(s[1:], t[1:])`). Funkcija tedaj ne more biti nič drugega kot

```
def enaka(s, t):
    return not s and not t or s and t and s[0] == t[0] and enaka(s[1:], t[1:])
```

148. Preverjanje Fibonacija

Spet le prevedemo navodilo v Python: seznam vsebuje Fibonacijevo zaporedje, če ima manj kot tri elemente (`len(s) < 3`) ALI pa je zadnji element vsota predzadnjih dveh (`s[-1] == s[-2] + s[-3]`) in je tudi seznam brez zadnjega elementa Fibonacijev (`je_fibo(s[:-1])`).

```
def je_fibo(s):
    return len(s) < 3 or (s[-1] == s[-2] + s[-3]) and je_fibo(s[:-1])
```

Lahko pa gremo tudi od spredaj.

```
def je_fibo(s):
    return len(s) < 3 or (s[2] == s[0] + s[1]) and je_fibo(s[1:])
```

149. Naraščajoči seznam

Seznam je naraščajoč, če ima manj kot dva elementa ali če je prvi element manjši od drugega in je naraščajoč tudi seznam od drugega elementa naprej.

```
def narascajoci(s):
    return len(s) < 2 or s[0] < s[1] and narascajoci(s[1:])
```

150. Vsota gnezdenega seznama

Naloga, za razliko od palindromov in iskanja elementa in vsote seznama števil, praktično *zahteva* rekurzijo – brez nje bi nam bilo težje. Pri tem lahko uporabimo rekurzijo samo tam, kjer je potrebujemo ali pa tudi tam, kjer bi šlo brez nje.

Najprej tam, kjer jo potrebujemo.

```
def vsota2(s):
    vsota = 0
    for x in s:
        if isinstance(x, list):
            vsota += vsota2(x)
        else:
            vsota += x
    return vsota
```

Z zanko gremo prek seznama. Za vsak element pogledamo, če gre za (vgnezdeni) seznam. Če je tako, rekurzivno pokličemo funkcijo `vsota2`, da vrne vsoto za ta podseznam, in rezultat prištejemo k skupni vsoti. Če pa element ni seznam, ga le prištejemo.

Zdaj pa naredimo reč bolj rekurzivno. V rešitvah prejšnjih nalog smo videli, da smo se z rekurzijo pravzaprav znebili zanke. Tudi tu se je dajmo. Tako kot v nalogi, kjer smo računali vsoto seznama števil, tudi tu recimo: vsota je enaka vsoti prvega elementa in vsoti ostanka seznama. Razlika je le v tem, da je ta, prvi element, zdaj lahko število ali seznam.

```
def vsota2(s):
    if not s:
        return 0
    if isinstance(s[0], list):
        return vsota2(s[0]) + vsota2(s[1:])
    else:
        return s[0] + vsota2(s[1:])
```

Lepo, ne? Če je `s[0]` število, ga prištejemo kar tako, če seznam, ga spustimo prek `vsota2`. Reč lahko obrnemo tako, da `vsota2` sprejme tudi število in v tem primeru vrne kar to število.

```
def vsota2(s):
    if not isinstance(s, list):
        return s
    if not s:
        return 0
    return vsota2(s[0]) + vsota2(s[1:])
```

Funkcijo – v prvi ali v drugi obliki – se da še nekoliko skrajšati, a potem ni več prav berljiva.

151. Obrni

Če je niz prazen, je tak tudi obrnjeni niz. Sicer pa naredimo, kot pravi naloga: k obrnjenemu nizu brez prve črke prištejemo prvo črko.

```
def obrni(s):
    if not s:
        return ""
    return obrni(s[1:]) + s[0]
```

Lahko naredimo tudi obratno: niz obrnemo tako, da k zadnji črki prištejemo obrnjen niz brez zadnje črke.

```
def obrni(s):
    if not s:
        return ""
    return s[-1] + obrni(s[:-1])
```

Še ena simpatična bližnjica:

```
def obrni(s):
    return s and s[-1] + obrni(s[:-1])
```

Če je `s` prazen, je neresničen. Python v tem primeru ne preverja teag, kar sledi `and`-u in vrne `s`, torej prazen niz. Če je `s` neprazen, pa preveri, kaj je desno od `and`-a in vrne to.

152. Zrcalo

Zrcalo lahko dobimo tako, da k nizu prištejemo `|` in še obrnjeni niz, torej

```
def zrcalo(s):
    return s + "|" + obrni(s)</xmp>
```

Obračanje niza pa smo napisali v prejšnji nalogi.

Vendar je to nalogo veliko zabavneje reševati tako, da je rekurzivna kar tale funkcija. Takole:

```
def zrcalo(s):
    if not s:
        return "|"
    return s[0] + zrcalo(s[1:]) + s[0]
```

Prazen niz zrcalimo tako, da vrnemo le `|`. Če niz ni prazen, pa bo imel "zrcaljeni niz" na začetku in na koncu prvi znak, vmes pa prezrcaljeni ostanek niza. Torej, `"janez"` prezrcalimo tako, da med `"j"` in `"j"` postavimo prezrcaljeni `"anez"`.

153. Kam?

Nanizali bomo kar nekaj rešitev, od preprostih do zanimivih.

Prva ni nič posebnega: `korak`amo po poti. Recimo, da smo v sobi `soba`; ko vidimo L, stopimo v `zemljevid[soba][0]`, ko D, v `zemljevid[soba][1]`.

```
def prehodi_pot(zemljevid, soba, pot):
    for korak in pot:
        if korak == "L":
            soba = zemljevid[soba][0]
        else:
            soba = zemljevid[soba][1]
    return soba
```

Python ima nek grozni izraz `if-else`. Uporabimo ga lahko, da povemo, da je naslednja soba `zemljevid[soba][0]`, če je `korak == "L"`, sicer pa `zemljevid[soba][0]`. Ali, z istimi besedami v Pythonu:

```
def prehodi_pot(zemljevid, soba, pot):
    for korak in pot:
        soba = zemljevid[soba][0] if korak == "L" else zemljevid[soba][1]
    return soba
```

Če pogledamo pozorneje, vidimo, da gremo vedno v `zemljevid[soba][nekaj]`, pri čemer je

`nekaj` `0`, če je `korak == "L"`, sicer pa `1`.

```
def prehodi_pot(zemljevid, soba, pot):
    for korak in pot:
        soba = zemljevid[soba][0 if korak == "L" else 1]
    return soba
```

To pa je kajpak nesmiselno: rečemo lahko preprosto `korak == "D"`. Ta izraz, `korak == "D"`, ima vrednost `False` (kar je isto kot `0`), kadar je `korak` enak `"L"` in `True` (kar je isto kot `1`), kadar je `korak` enak `"D"`.

```
def prehodi_pot(zemljevid, soba, pot):
    for korak in pot:
        soba = zemljevid[soba][korak == "D"]
    return soba
```

Najlepša rešitev te naloge pa je rekurzivna. Najprej jo zapišimo nekoliko daljše.

```
def prehodi_pot(zemljevid, soba, pot):
    if pot:
        return prehodi_pot(zemljevid, zemljevid[soba][pot[0] == "D"], pot[1:])
    else:
        return soba
```

Če je pot neprazna, funkcija vrne tisto sobo, ki jo dobi z (novim) klicem same sebe, pri čemer kot argument da naslednjo sobo in `pot` brez prvega znaka. Če pa je `pot` prazna, vrne trenutno sobo.

Celotno funkcijo lahko – po zgledu drugih rekurzivnih funkcij, ki smo jih napisali – skrajšamo v en sam izraz.

```
def prehodi_pot(zemljevid, soba, pot):
    return pot and prehodi_pot(zemljevid, zemljevid[soba][pot[0] == "D"],
pot[1:]) or soba
```

Izraz ne bi delal dobro, kadar bi rekurzivni klic funkcije `prehodi_pot` vrnil `0`. Tako, kot so sestavljeni naši zemljevidi, se to ne zgodi. Če bi se tega bali, pa napišemo, kot je bolj prav:

```
def prehodi_pot(zemljevid, soba, pot):
    return prehodi_pot(zemljevid, zemljevid[soba][pot[0] == "D"], pot[1:]) if pot
else soba
```

154. Sodi – lihi – rekurzivno

Kot je pri pisanju rekurzivnih funkcij pogosto, moramo tudi tokrat le prevesti definicijo iz slovenščine v Python (če je definicija rekurzivna, seveda).

- Seznam je sodo-lih, če je prazen, ali pa je prva številka soda, ostanek pa je liho-sod (torej: začne se z liho in potem se izmenjujejo lihe in sode).
- Seznam je liho-sod, če je prazen, ali pa je prva številka liha, ostanek pa je sodo-lih (torej: začne se s sodo in potem se izmenjujejo lihe in sode).

```
def sodi_lihi(s):
    return not s or s[0] % 2 == 0 and lihi_sodi(s[1:])

def lihi_sodi(s):
    return not s or s[0] % 2 == 1 and sodi_lihi(s[1:])
```

155. Razdalja do cilja

Tole je pravzaprav isto kot "Kako daleč", le da smo tam iskali potomca z določenim imenom, tu sobo z določeno številko.

Zmenili se bomo takole: če pod trenutno sobo ni sobe 42, naj funkcija vrne `None`. Naloga je postavljena tako, kot da soba 42 vedno obstaja, zato o tem, kaj naj funkcija vrne, kadar te sobe ni, niti ne govori. V rekurzivnem klicu pa bomo to potrebovali: ko bomo poklicali funkcijo za neko sobo v nižjem nadstropju, bo funkcija vrnila bodisi globino sobe 42, bodisi `None`, ki nam bo povedal, da moramo iskati drugje.

Zdaj, ko smo se dogovorili o tem, je reč preprosta. Če smo že v sobi 42, vrnemo globino 0 (`if soba == 42: return 0`. Sicer pogledamo sobi na obeh straneh (`for spodaj in zemljevid[soba]`). Če spodnja soba obstaja (`if spodaj is not None` - enako dobro bi bilo tudi `if spodaj`, pa tudi `if spodaj != None` bi delovalo), potem pogledamo, kako globoko pod to, spodnjo sobo, je prstan (`g = globina_prstana(zemljevid, spodaj)`). Če funkcija vrne `None`, pod to spodnjo sobo ni prstana. Vendar nas to ne zanima, zanima nas primer, ko funkcija ne vrne `None`, temveč globino (`if g is not None`). V tem primeru je od trenutne sobe do prstana toliko, kolikor je bilo od spodnje sobe do prstana in še 1 zraven (`return 1 + g`), saj moramo prišteti še korak v spodnjo sobo.

```
def globina_prstana(zemljevid, soba):
    if soba == 42:
        return 0
    for spodaj in zemljevid[soba]:
        if spodaj is not None:
            g = globina_prstana(zemljevid, spodaj)
            if g is not None:
                return 1 + g
```

Zanka `for` gre čez obe spodnji sobi, na podoben način, kot smo pri preiskovanju rodbine celjskih grofov delali zanko prek otrok. Če ne na eni ne na drugi strani ne najde ničesar, tudi ne vrne ničesar, torej vrne `None`.

156. Pot do cilja

Tudi tole smo videli že pri rodbini, le da smo tam sestavljali seznam otrok, ki vodi do določenega potomca, tule pa niz L-jev in D-jev.

Po drugi strani je funkcija las podobna funkciji za izračun globine. Razlikujeta se le po tem, kaj vračata: ena vrača globino, druga pot. Če smo že v sobi 42, smo prej vrnili 0, zdaj prazno pot, `""`. Če se prek spodnje sobe pride do prstana, smo prej vrnili globino, ki smo ji prišteli 1, zdaj vrnemo pot, ki ji prištejemo korak, ki vodi v spodnjo sobo.

Zanko prek spodnjih sob napišemo malenkost drugače kot prej: namesto `for spodaj in zemljevid[soba]` zdaj pišemo `for spodaj, smer in zip(zemljevid[soba], "LD")`, da dobimo številko spodnje sobe (ta bo v spremenljivki `spodaj`) in še korak, ki vodi vanjo, `"L"` ali `"D"` (ta bo v spremenljivki `smer`).

Če v zanki ne najdemo nobene poti, ki bi vodila v iskano sobo, ne vrnemo ničesar (torej vrnemo `None`). Če pa najdemo pot iz ene od spodnjih sob do cilja, pa vrnemo to pot, spredaj pa pripnemo še smer, ki vodi v spodnjo sobo.

```
def pot_do_prstana(zemljevid, soba):
    if soba == 42:
        return ""
    for spodaj, smer in zip(zemljevid[soba], "LD"):
        if spodaj is not None:
            pot = pot_do_prstana(zemljevid, spodaj)
            if pot is not None:
                return smer + pot
```

Če imamo to funkcijo, bi lahko funkcijo za izračun dolžine poti sestavili kar tako, da bi vrnili ... no, dolžino poti pač.

```
def globina_prstana(zemljevid, soba):
    return len(pot_do_prstana(zemljevid, soba)
```

157. Naprej nazaj

Spet le prepišemo definicijo iz naloge v rekurzivno funkcijo. Če si pomagamo še s slovarjem, ki ga je priporočila naloga, je še lažje: žaba gre naprej in nazaj, če je njena pot dolga 0 ali pa je prvi znak obraten zadnjemu ("S" namesto "J" in podobno) ter zaporedje med prvim in zadnjim znakom prav tako predstavlja pot nekam in nazaj.

```
def naprej_nazaj(s):
    obratno = {"S": "J", "J": "S", "V": "Z", "Z": "V"}
    return len(s) == 0 or s[-1] == obratno[s[0]] and naprej_nazaj(s[1:-1])</xmp>
```

Je to gotovo pravilno? Kaj se zgodi, ko če dobimo niz dolžine 1? Lahko tej primerjamo prvi in zadnji element? Tudi takrat funkcija naredi natančno, kar mora: prvi in zadnji element sta tedaj en in isti element. V tem primeru pogoj, da mora biti prvi obratni kot zadnji, ne more biti izpolnjen in funkcija bo (kot tudi mora) za vse nize dolžine 1 vrnila `False`.

158. Aritmetično zaporedje

Naloga pravi, da je zaporedje aritmetično, če ima največ dva elementa (`len(s) <= 2`) ali pa je razlika med prvim dvema enaka razliki med drugim in tretjim (`s[1] - s[0] == s[2] - s[1]`) in je aritmetičen tudi ostanek zaporedja (`aritmeticno(s[1:])`). Torej mora biti rešitev naloge

```
def aritmeticno(s):
    return len(s) <= 2 or s[2] - s[1] == s[1] - s[0] and aritmeticno(s[1:])
```

159. Binarno

Če je število enako 0 ali 1, vrnemo to število, pretvorjeno v niz. Sicer pa število delimo z 2 in ga spremenimo v dvojiško, na konec pa pripnemo zadnjo števko - dobimo jo kot ostanek po deljenju števila z 2.

```
def binarno(n):
    if n <= 1:
        return str(n)
    return binarno(n // 2) + str(n % 2)
```

Kdor rad packa, naredi tako.

```
def binarno(n):
    return (n > 1 and binarno(n // 2) or "") + str(n % 2)
```

160. Decimalno

Če je niz dolg 1, torej, če je enak `"0"` ali `"1"`, ga le pretvorimo iz niza v število, pa smo opravili. Sicer pa pretvorimo v dvojiški zapis vse razen zadnje števke, to pomnožimo z 2 in prištejemo zadnjo števko.

```
def decimalno(s):
    if len(s) == 1:
        return int(s)
    return 2 * decimalno(s[:-1]) + int(s[-1])
```

Ljudje zvitih misli naredijo tole:

```
def decimalno(s):
    return 2 * int(len(s) > 1 and decimalno(s[:-1])) + int(s[-1])
```

161. Zadnje liho

Naloga je zanimiva, ker bi si vsak zrel programer predstavljal tole rešitev.

```
def zadnje_liho(s):
    if not s:
        return None
    t = zadnje_liho(s[1:])
    if not t and s[0] % 2 == 1:
        t = s[0]
    return t
```

Če je seznam prazen, lihih števil ni in vrnemo `None`. Sicer poiščemo zadnje liho v preostanku seznama. Če ga tam ni, vendar je liho kar prvo število, je očitno to zadnje liho število.

Krajše (a ne preveč lepo) je tako:

```
def zadnje_liho(s):
    return s and (zadnje_liho(s[1:]) or s[0] % 2 and s[0]) or None
```

Tako bi torej naredil vsak zrel programer, ki je že kdaj delal v *funkcijskih jezikih*, kjer imamo pogosto neposreden dostop le do prvega, ne pa tudi do zadnjega elementa seznama. Študenti, ki so to nalogo reševali na izpitu, so to obrnili drugače: če iščemo s konca, pač iščimo s konca.

```
def zadnje_liho(s):
    if not s:
        return None
    if s[-1] % 2 == 1:
        return s[-1]
    return zadnje_liho(s[:-1])
```

Tudi prav. To je preprostejše in v Pythonu povsem legalno.

162. Indeksi

Če je seznam prazen, vrnemo prazen seznam indeksov. Sicer pogledamo zadnji element. Če je enak iskanemu, bomo vrnili indekse tega elementa v vsem seznamu brez tega, zadnjega elementa, in še ta element zraven. Sicer pa vrnemo pač le indekse elementov v vseh elementih razen zadnjega.

```
def indeksi_rec(s,e):
    if not s:
        return []
    if s[-1] == e:
        return indeksi_rec(s[:-1], e) + [len(s)-1]
    else:
        return indeksi_rec(s[:-1], e)
```

Če gre komu na živce, da dvakrat ponavljamo `indeksi_rec(s[:-1], e)`, ga lahko potolažimo s skrajšano različico (ki je ne bomo komentirali: kogar zanimajo takšne perverznosti, naj sam uživa v raziskovanju):

```
def indeksi_rec(s, e):
    return s and (indeksi_rec(s[:-1], e) + [len(s) - 1] * (e == s[-1]))
```

Zadaj pa poskusimo še od leve proti desni, kot rekurzija teče običajno. Ta rešitev je bolj zapletena:

```
def indeksi_rec(s, e):
    if not s:
        return []
    ost = [x + 1 for x in indeksi_rec(s[1:], e)]
    if s[0] == e:
        return [0] + ost
    else:
        return ost
```

Stvar je podobna kot z zadnjim elementom, le da gre v rekurzivni klic seznam brez prvega elementa, zato so vsi indeksi, ki jih vrne, za 1 premajhni. To rešimo tako, da jim enico prištejemo.

Tudi to rešitev se da obrniki v eno vrstico.

```
def indeksi_rec(s, e):
    return s and [0] * (s[0] == e) + [x + 1 for x in indeksi_rec(s[1:], e)]
```

163. Brez jecljanja

Če sta prva dva znaka enaka, prvega enostavno ignoriramo; rezultat funkcije je enak nejecljajočemu ostanku seznama. Če nista enaka, pa vzamemo prvi znak (kot seznam) in dodamo nejecljajoči ostanek seznama.

Rekurzija se konča, ko imamo le še en znak.

```
def brez_jecljanja_rec(s):
    if len(s) < 2:
        return s
    if s[0] == s[1]:
        return brez_jecljanja_rec(s[1:])
    else:
        return [s[0]] + brez_jecljanja_rec(s[1:])
```

Namesto `[s[0]]` lahko pišemo tudi `s[:1]`. `s[0]` pa ne bo delovalo, ker je to prvi element seznama (in ne seznam, ki vsebuje prvi element seznama), zato k njemu ne morem prišteti preostanka seznama.

Bližnjica za vse to je

```
def brez_jecljanja_rec(s):
    return s if len(s) < 2 else s[:s[0] != s[1]] + brez_jecljanja_rec(s[1:])
```

Izraz `s[0] != s[1]` je resničen ali ne, torej je 0 ali 1. Torej je `s[:s[0] != s[1]]` bodisi `s[:0]` (prvih `0` elementov `s`-a) ali `s[:1]` (seznam, ki vsebuje prvi element `s`-a). Ostalo je očitno.

164. Rekurzivni Collatz

Če je `n` enak `1`, vrnemo `1`. Sicer pa vrnemo za 1 več, kot je dolgo zaporedje od naslednjega člena naprej.

```
def collatz(n):
    if n == 1:
        return 1
    elif n % 2 == 0:
        return 1 + collatz(n // 2)
    else:
        return 1 + collatz(n * 3 + 1)
```

165. Sprazni

Le pomoči sledimo: gremo čez seznam. Sestavimo nov seznam. Če vidimo kaj, kar ni `None`, torej seznam, vstavimo v novi seznam izpraznjeni seznam.

```
def sprazni(s):
    novi = []
    for e in s:
        if e is not None:
            novi.append(sprazni(e))
    return novi
```

166. Rekurzivni štumfi, zokni, kalcete, fuzetlne in kucjte

Če je seznam nogavic prazen, podana nogavica ni brez para in vrnemo `False`.

Sicer preverimo, ali je prva nogavica enaka podani. Če je, bo brez para, če v ostanku seznama ta nogavica *ni* brez para. Sicer pa prvi element ignoriramo in gledamo ostanek seznama.

```
def brez_para(stevilka, nogavice):
    if not nogavice:
        return False
    if nogavice[0] == stevilka:
        return not brez_para(stevilka, nogavice[1:])
    else:
        return brez_para(stevilka, nogavice[1:])
```

Ker gre za same `if`-e in `return`-e, lahko to brez težav povemo v eni vrstici. Le nekaj zabavnega potrebujemo v drugem pogoju. Ga razumeš?

```
def brez_para(stevilka, nogavice):
    return nogavice != [] \
        and (nogavice[0] == stevilka) != brez_para(stevilka, nogavice[1:]
```

Splošne vaje iz programiranja

167. Stopnice

V `v` bomo shranjevali višino, do katere je robot priplezal. Nato gremo po stopnicah in na vsakem koraku preverimo, ali je stopnica previsoka (`e - v > 20`). V tem primeru končamo vzpon (`break`). Če stopnica ni previsoka, robot stopi nanjo (`v = e`). Če se zanka konča z `break`, bo v `v` višina zadnje stopnice, do katere je robot prišel. Če se konča "po naravni poti", torej, ker je zmanjkalo stopnic, bo v `v` višina zadnje stopnice, kar je spet prav.

```
def kako_visoko(stopnice):
    v = 0
    for e in stopnice:
        if e - v > 20:
            break
        v = e
    return v
```

Za tiste, ki jih zanima: naloga ima tudi lepo rešitev v eni vrstici.

```
from functools import reduce

def kako_visoko(stopnice):
    return reduce(lambda x, y: x if y - x > 20 else y, stopnice, 0)
```

168. Drugi največji element

Naloga predstavlja duhamorno, a koristno telovadbo.

```
xs = [5, 1, 4, 8, 2, 3]
prvi = drugi = float('-inf')
for x in xs:
    if x > prvi:
        drugi = prvi
        prvi = x
    elif x > drugi:
        drugi = x
print(drugi)
```

Lepota rešitve je v tem, da pove v Pythonu natančno to, kar bi s praktično istimi besedami povedali neformalno. V zanki pravimo: preglej vse elemente in če je trenutni element seznama (`x`) večji od največjega (`prvi`), bo trenutni postal največji, ta, ki je bil doslej `prvi`, pa bo poslej `drugi`. Pri tem je pomembno, da prirejanji napišemo v pravem vrstnem redu; če bi pisali

```
        prvi = x
        drugi = prvi
```

to ne bi bilo najbolj posrečeno, saj bi `prvi` in `drugi` postala enaka `x`.

Temu sledi `elif`, *sicerče*: če `x` sicer ni večji od največjega, je pa večji od drugega največjega, je `x` novi `drugi` največji.

Pred zanko si privoščimo mali tehnični trik: `prvi` in `drugi` element naj bosta minus neskončno; dobimo ga s `float('-inf')`. Tako zagotovimo, da bo vse, na kar bomo naleteli, večje od njiju.

Nekoliko iznajdljivejša rešitev naroči Pythonu, naj uredi seznam, in izpiše drugi največji element. Rešitev celotne naloge je tako

```
print(sorted(xs)[-2])
```

Takšna rešitev ni povsem v skladu z navodili, saj bo v primeru, da imamo dva enaka največja elementa (denimo dve osmici) izpisala njuno vrednost. Rešimo se lahko tako, da seznam spremenimo v množico; v njej se vsak element pojavi le enkrat.

```
print(sorted(set(xs))[-2])
```

Mimogrede omenimo, da lahko urejanje seznama vzame precej časa, če je seznam dolg, množica pa lahko potrati precej pomnilnika. A pri nalogicah, kot so te, ne bo krize. Če boste nalogo kdaj reševali zares in boste potrebovali, recimo, največjih `k` število iz ogromnega kupa z *n* števili, pa takrat, ko vas bodo učili o podatkovnih strukturah, ne prespite poglavja o kopicah.

169. Collatz 2

Ta naloga pa zahteva bodisi zanko znotraj zanke bodisi funkcijo.

```
naj_st = naj = 1
for i in range(1, 100001):
    korakov = 1
    n = i
    while n != 1:
        if n % 2 == 0:
            n //= 2
        else:
            n = n * 3 + 1
        korakov += 1
    if korakov > naj:
        naj = korakov
        naj_st = i
print(naj_st, naj)
```

Zunanja zanka teče prek števil od 1 do 10000. Notranja zanka izračuna zaporedje za vsako od njih in prešteva korake. V if-u, ki ji sledi, preverimo, ali je to zaporedje daljše od najdaljšega doslej. Če je, si zapomnimo tako njegovo dolžino kot število, s katerim smo ga dobili.

Program postane bistveno preglednejši, če izračun dolžine zaporedja zapremo v funkcijo.

```
def collatz(n):
    korakov = 1
    while n != 1:
        if n % 2 == 0:
            n //= 2
        else:
            n = n * 3 + 1
        korakov += 1
    return korakov

naj_st = naj = 1
for i in range(1, 100001):
    korakov = collatz(i)
    if korakov > naj:
        naj = korakov
        naj_st = i
print(naj_st, naj)
```

Čeprav imamo *v bistvu* še vedno zanko v zanki, sta na pogled ločeni: ena računa dolžino zaporedja, druga preskuša različna števila. Medtem ko prvi program še resen programer težko prebere – začetnik pa še težje – je drugi sestavljen iz dveh preprostih kosov in zato pregleden.

170. Delnice

Za probleme tega tipa obstajajo učinkovitejše rešitve, ki pa ne sodijo v osnovni tečaj programiranja; zanj je preprosta rešitev z dvema zankama čisto primerna. V zunanji zanki poskušamo mesece nakupa delnice, za vsak mesec nakupa pa poskusimo vse možne mesece prodaje. Dobiček izračunamo tako, da seštejemo spremembe vrednosti delnice od meseca nakupa do meseca prodaje.

```
def posrednik(delnica):
    naj_dobicek = -1
    for od in range(12):
        for do in range(od, 12):
            dobicek = sum(delnica[od:do])
            if dobicek > naj_dobicek:
                naj_dobicek = dobicek
                naj_od, naj_do = od, do
    return naj_od, naj_do
```

Program lahko naredimo malenkost učinkovitejši (in poučnejši), če se znebimo `sum` in seštevamo sami. Način, na katerega uporabljamo `sum`, je, kakor da bi se iz računalnika norčevali. Sprašujemo ga reči kot "*koliko je vsota od petega do osmega elementa*", v naslednjem krogu "*koliko je vsota od petega do devetega*" in v naslednjem, "*koliko je vsota od petega do desetega elementa*" ... očitno toliko kot prej, le še deseti element prištejemo, ne?

```
def posrednik(delnica):
    naj_dobicek = naj_od = naj_do = 0
    for od in range(12):
        dobicek = 0
        for do in range(od, 12):
            dobicek += delnica[do]
            if dobicek > naj_dobicek:
                naj_dobicek = dobicek
                naj_od, naj_do = od, do
    return naj_od, naj_do+1
```

V programu se ne držimo povsem Pythonovih pravil indeksiranja: v `dobicek` so vključene vrednosti od `od` do *vključno* `do`. Da bi kljub temu dobili pravilni interval, kot ga pričakuje naloga, smo na koncu k `naj_do` prišteli 1, tako da gornja meja ni vključena v interval.

171. Spremembe smeri

Z `i` bomo šteli od 2 naprej in primerjali razliko tem elementom in predzadnjim (s`[i]` in `s[i-1])` predzadnjim in predpredzadnjim (`s[i-1]` in `s[i-2]`). Če imata različen predznak, bomo to šteli kot spremembo.

Kako preverimo, ali imata števili različen predznak? Lahko si napišemo funkcijo

```
def sign(x):
    if x > 0:
        return 1
    elif x < 0:
        return -1
    else:
        return 0
```

ali kaj podobnega. Potem bomo različnost predznakov izmerili z

```
sign(s[i] - s[i - 1]) != sign(s[i - 1] - s[i - 2])
```

Če nočemo takšne funkcije, si lahko grenimo življenje (in delamo napake v programih) z

```
s[i]-s[i-1] > 0 and s[i-1]-s[i-2] < 0 or s[i]-s[i-1] < 0 and s[i-1]-s[i-2] > 0
```

Lahko pa se domislimo trika s produktom: razliki zmnožimo. Če sta imeli različen predznak, bo produkt negativen, če enakega, pozitiven.

```
(s[i] - s[i - 1]) * (s[i - 1] - s[i - 2]) < 0
```

Tako je z najtežjim delom naloge opravljeno, napisati je potrebno le še vse ostalo.

```
def sprememb_smeri(s):
    sprememb = 0
    for i in range(2, len(s)):
        if (s[i - 2] - s[i - 1]) * (s[i - 1] - s[i]) < 0:
            sprememb += 1
    return sprememb
```

172. Sekajoči se krogi

Najprej se spomnimo, kako ugotoviti, ali se dva kroga sekata: izračunati moramo razdaljo med njunima središčema in jo primerjati z vsoto njunih polmerov. Če je razdalja med središčema manjša, se kroga sekata.

Naivna rešitev programa je takšna.

```
from math import sqrt

def sekajo(krogi):
    for x1, y1, r1 in krogi:
        for x2, y2, r2 in krogi:
            if sqrt((x1 - x2)**2 + (y1 - y2)**2) <= r1 + r2:
                return True
    return False
```

Spet imamo dvojno za zanko: za vsak krog preverimo vse kroge, da vidimo ali se kak par seka. Razmislite, kako bi nalogo reševali ročno, pa boste videli, da je isto.

Vzorec je spet tak kot v praštevilih, unikatna težava naloge pa je v tem, da se vsak krog seka sam s sabo. Rešitve z dodatnim pogojem

```
if not (x1 == x2 and y1 == y2 and r1 == r2)
```

ne maramo, saj imamo lahko slučajno v seznamu dva enaka kroga.

Kroge lahko oštevilčimo, recimo z `enumerate`, in ne preverjamo sekanja med krogoma z isto številko, saj tedaj vemo, da gre za isti krog.

```
def sekajo(krogi):
    for i, (x1, y1, r1) in enumerate(krogi):
        for j, (x2, y2, r2) in enumerate(krogi):
            if i != j and sqrt((x1 - x2)**2 + (y1 - y2)**2) <= r1 + r2:
                return True
    return False
```

Bodite pozorni, kako v razpakiramo oštevilčene terke: imamo številko in terko, torej dve stvari, saj enumerate vedno vrača pare. Razpakirati ju moramo torej v dve stvari, številka gre v `i`, drugi element pa je terka s tremi elementi, ki jo spet razpakiramo.

Mimogrede, če bi šlo za krožnice in ne kroge, bi morali popaziti na primer, ko je en krog (oziroma krožnica) v celoti znotraj drugega. V tem primeru bi morala biti razdalja med središčema manjša od vsote polmerov, a manjša od absolutne vrednosti njune razlike (nariši, pa boš videl, zakaj!), `abs(r1 - r2) <= sqrt((x1 - x2)**2 + (y1 - y2)**2) <= r1 + r2`.

Nadaljujmo z izboljševanjem rešitve. Če malo premislimo, ni nobene potrebe, da bi primerjali vsak krog `i`-ti krog z vsemi drugimi: zadošča, da ga primerjamo z vsemi pred `i`-tim. Tako bo program dvakrat hitrejši, saj naredi dvakrat manj primerjav, pa še problem sekanja kroga s samim sabo rešimo.

```
def sekajo(krogi):
    for i, (x1, y1, r1) in enumerate(krogi):
        for x2, y2, r2 in krogi[:i]:
            if sqrt((x1 - x2)**2 + (y1 - y2)**2) <= r1 + r2:
                return True
    return False
```

Kar vidimo tu – dvojna zanka, pri čemer notranja teče le do tam, do kjer je prišla zunanja – je pogosta reč. Čeprav začetnika v Pythonu ne želimo navajati na `range(len(s))`, tokrat naredimo izjemo, da bo trik lažje prenesti v druge jezike z drugačnimi zankami `for`. Tole je rešitev s čistim `range`-`len`.

```
def sekajo(krogi):
    for i in range(len(krogi)):
        for j in range(i):
            x1, y1, r1 = krogi[i]
            x2, y2, r2 = krogi[j]
            if sqrt((x1 - x2)**2 + (y1 - y2)**2) <= r1 + r2:
                return True
    return False
```

Lahko bi se znebili spremenljivk `x1`, `y1` in tako naprej ter v formuli pisali `(krogi[i][0] - krogi[j][0])**2` in tako dalje. Ne počnite tega, nepregledno je. Obstaja tudi vmesna varianta, v kateri zanke pišemo takole:

```
    for i in range(len(krogi)):
        for x2, y2, r2 in krogi[:i]:
            x1, y1, r1 = krogi[i]
```

Za konec pa še ena zanimiva rešitev. Rekli smo, da krogov ne smemo primerjati, saj lahko naletimo na dva *enaka* kroga. To drži, vendar ima Python tudi operator, ki preverja *istost*.

```
def sekajo(krogi):
    for krog1 in krogi:
        for krog2 in krogi:
            if krog1 is krog2:
                continue
            x1, y1, r1 = krog1
            x2, y2, r2 = krog2
            if sqrt((x1 - x2)**2 + (y1 - y2)**2) <= r1 + r2:
                return True
    return False
```

Ta rešitev nima nobenih prednosti pred prejšnjimi in še počasnejša je. Vseeno pa je zanimivo vedeti zanjo.

173. Največ n-krat

Upoštevali bomo namig: seznam ni dolg. Šli bomo prek seznama, dodajali njegove elemente v nov seznam, vendar predtem vedno prešteli, da jih ni že preveč.

```
def najvec_n(s, n):
    novi = []
    for e in s:
        if novi.count(e) < n:
            novi.append(e)
    return novi
```

Pri dolgih seznamih bi utegnil postati program počasen zaradi funkcije `count`, ki mora tolikokrat preštevati elemente (vedno daljšega seznama `novi`). Boljše bi bilo, ko bi si zabeležili, koliko pojavitev vsakega elementa smo že dodali vanj. Za to bi uporabili slovar. S slovarji in preštevanjem se ukvarjajo naloge v ločenem razdelku, tule le pokažimo rešitev za tiste, ki to že znajo.

```
import collections
def najvec_n(s, n):
    novi = []
    pojav = collections.defaultdict(int)
    for e in s:
        if pojav[e] < n:
            novi.append(e)
        pojav[e] += 1
    return novi
```

Ta različica je hitra, a manj splošna, saj deluje le, če so elementi podanega seznama nespremenljivi. Dokler vsebuje seznam le števila in nize (pa še par podobno preprostih tipov), smo varni; če bi vseboval sezname, pa funkcija ne bi več delovala, saj seznami ne morejo biti ključi slovarjev.

Lotimo se še rešitve z brisanjem, saj bo pokazala pogosto začetniško napako. Naivec bi napisal tako (izpustil bi le komentar v prvi vrstici).

```
# Ta program ne deluje!
def najvec_n(s, n):
    novi = s[:]
    for i in range(len(novi)):
        if novi[:i].count(novi[i]) == n:
            del novi[i]
```

V začetku naredimo kopijo seznama, saj nočemo spreminjati originala. Z `i`-jem se sprehodimo od začetka do konca seznama. Ko se odločamo o tem, ali pobrisati `i`-ti element (njegova vrednost je `novi[i]`), preverimo, kolikokrat se le-ta pojavi v doslej pregledanem delu seznama, torej v `novi[:i]`). Če je `novi[:i].count(novi[i])` že enak `n`, je potrebno ta element pobrisati, saj je takšnih že dovolj.

Problem pa je tule. Na primer, da ima seznam v začetku 15 elementov. Števec i se bo junaško lotil prehoditi `range(len(novi))`, torej `range(15)`. Žal pa se seznam `novi` med izvajanjem

funkcije krajša in še preden bo `i` prilezel do konca, do 14, bo padel prek gornje meje seznama. Bolj domače povedano, seznam se mu bo izpodmaknil.

Naivec vztraja. Zanko `for` zamenja z while, ki vsakič sproti preveri, ali je `i` še dovolj majhen (oz. povedano z druge strani, ali je seznam še dovolj dolg).

```
# Tudi ta program ne deluje
def najvec_n(s, n):
    novi = s[:]
    i = 0
    while i < len(novi):
        if novi[:i].count(novi[i]) == n:
            del novi[i]
        i += 1
    return novi
```

Da program ne deluje, se prepričamo kar na primeru iz naloge: iz seznama `[1, 2, 3, 1, 1, 2, 1, 2, 3, 3, 2, 4, 5, 3, 1]` bo naredil `[1, 2, 3, 1, 1, 2, 2, 3, 3, 4, 5, 1]`: enica se pojavi štirikrat, čeprav dopuščamo le tri ponovitve. Še očitneje gre narobe, če pokličemo `najvec_n([1, 1, 1, 1, 1, 1, 1, 1], 3)`. Rezultat je klavrn: `[1, 1, 1, 1, 1, 1, 1]`.

Napaka je prefinjena in vsak zaresen programer jo je parkrat zagrešil, odtlej pa pazi nanjo še bolj kot na deljenje z nič. Denimo, da je `i` enak `7` in da je sedmi element potrebno pobrisati. Funkcija torej reče `del novi[i]` in nadaljuje z `i += 1`, s čimer `i` postane `8`. Osmi element nam je ušel! Ko smo pobrisali sedmega, je vskočil na njegovo mesto osmi. Števca `i` ne bi smeli povečati, ostati bi moral `7`, da bo preveril prejšnjega osmega, ki je po novem sedmi. Seznam se še vedno izpodmika, pogoj v zanki `while` nas zavaruje le pred prevelikimi indeksi.

Pravilna rešitev je torej

```
def najvec_n(s, n):
    novi = s[:]
    i = 0
    while i < len(novi):
        if novi[:i].count(novi[i]) == n:
            del novi[i]
        else:
            i += 1
    return novi
```

Ob tej navidez preprosti nalogi smo se zadržali kar dolgo. Kdor ne razume, zakaj predzadnji programi niso delovali, zadnji pa, si bo naredil veliko uslugo, če se bo poglobil v zadevo. Če tega ne stori zdaj, bo to napako kaj verjetno storil nekoč, ko bo šlo zares in ko jo bo težje odkriti, saj bo zakopana globoko v večjem programu.

Tudi različico rešitve, kjer brišemo, namesto da bi dodajali, je mogoče namesto s `count` izvesti s slovarji, tako kot smo naredili s prvo različico. A problem bi ostal enak.

174. Brez n-tih

Naloga preverja, ali smo se pri prejšnji nalogi kaj naučili. Če se nismo, napišemo

```
def brez_ntih(s, n):
    for i in range(n-1, len(s), n):
        del s[i]
```

In se čudimo, zakaj ne deluje. S tistim, ki sodi v to kategorijo, še enkrat premislimo, kaj se dogaja. Recimo, da želimo brisati vsak tretji element. V zanki `for i in range(0, len(s), n)` bo i enak 2, 5, 8 in tako naprej. Vendar se nam zgodi tole: ko pobrišemo drugi element, se peti element (ki ga prav tako želimo pobrisati!) prestavi na četrto mesto. Ko pobrišemo peti element, v resnici pobrišemo tistega, ki je bil prej šesti.

Včasih si pomagamo z brisanjem nazaj. Namesto da bi pobrisali elemente 2, 5 in 8, pobrišemo elemente 8, 5 in 2. Pri tem se nam nič ne spodmika in program deluje pravilno. Težave nam dela le izračun začetka brisanja. Takole: dolžino niza celoštevilsko delimo z n in pomnožimo z n. Tako pridemo do največjega večkratnika n, ki je še manjši od dolžine. (Kar prepričajmo se: 10//3*3 = 3*3 = 9; po drugi strani imamo 15//3*3 = 5*3 = 15.) Od tega odštejemo 1, saj ne brišemo elementov 3, 6, 9 temveč 2, 5, 8.

```
def brez_ntih(s, n):
    for i in range(len(s) // n * n - 1, 0, -n):
        del s[i]
```

Zdaj pa poskusimo boljše rešiti nalogo obrnjeno naprej: če hočemo pobrisati elemente 2, 5, 8, 11, se dogaja tole. Pobrišemo element 2. Elementi 5, 8 in 11 se premaknejo na 4, 7, 10. Da bi pobrisali element 5, bomo pobrisali element 4. Pri tem se 8 in 11, ki sta šla vmes na 7 in 10, premakneta na 6 in 9. Torej zdaj pobrišemo element 6 (da bomo pobrisali, kar je bil v začetku 8). Pri tem se začetni element 11 (ki je šel medtem na 10 in potem na 9) premakne na 8. Torej ga pobrišemo tako, da pobrišemo element 8. Kaj smo torej pobrisali? Elemente z indeksi 2, 4, 6 in 8! Hoteli smo brisati od elementa 2 s korakom 3, a ker se ob vsakem brisanju izpodmaknejo za en element, moramo brisati s korakom 2. Tokrat nam da vetra gornja meja. Dobimo jo lahko, recimo, tako kot v spodnjem programu.

```
def brez_ntih(s, n):
    for i in range(n - 1, len(s) - (len(s) + 1) // n + 1, n-1):
        del s[i]
```

S to nalogo so se sicer ubijali študenti na enem od izpitov. In niso bili preveč srečni, ko so videli, da je mogoče navodilo pobriši vsak n-1-vi element, začenši z elementom n-1 pravzaprav tudi dobesedno prevesti v Python:

```
def brez_ntih(s, n):
    del s[n-1::n-1]
```

175. Vse črke

Še pomnimo praštevila? Tole je isto: zanka, v kateri nekaj preverjamo in jo prekinemo, čim naletimo na to, kar iščemo (ali na tisto, česar nočemo).

```
def vse_crke(beseda, crke):
    for c in beseda:
        if not c in crke:
            return False
    return True
```

Ponovimo, marsikomu ne bo škodilo: `return True` mora biti izven zanke. Zelo narobe bi bilo

```
#Ta program ne deluje!
def vse_crke(beseda, crke):
    for c in beseda:
        if not c in crke:
            return False
        else:
            return True
```

Ta program bi vrnil True že, če bi bil zadovoljen s prvo črko. Zanka se namreč konča že v prvi iteraciji in že takoj vrne True ali False; druge črke ne pogleda nikoli.

Če poznamo množice, smo lahko še veliko krajši.

```
def vse_crke(beseda, crke):
    return not (set(beseda) - set(crke))
```

Izraz `set(beseda) - set(crke)` iz množice črk v besedi odvzame vse tiste, ki so v množici `crke`. Izraz `not (set(beseda)-set(crke))` vrne `True`, ko je ta množica prazna, kar pa je natanko takrat, ko so vse črke iz besede tudi v mnozici črk, oziroma, ko v množici črk besede ni nobene črke, ki ne bi bila tudi v množici črke.

Oklepaj pravzaprav ni potreben, saj ima odštevanje prednost pred `not`, vendar smo ga vseeno napisali. Čemu? Kako je videti tole:

```
def vse_crke(beseda, crke):
    return not set(beseda) - set(crke)
```

Vizualno sodi `not` k prvi množici in to ni dobro. Ko programiramo, se moramo potruditi, da nas videz programa ne bi varal.

176. Skritopis

Storili bomo tole: pripravili bomo nov niz in vanj znak za znakom prepisovali starega.

```
def skritopis(besedilo):
    novi = ""
    for znak in besedilo:
        novi += znak
    return novi
```

Ta funkcija sicer dela, vendar ne tistega, kar bi morala; le niz prepiše, namreč vse znake niza. Katere pa bi morala izpustiti? Vse tiste črke, pred katerimi je črka.

```
def skritopis(besedilo):
    novi = ""
    for znak in besedilo:
        if not znak.isalpha() or not novi or not novi[-1].isalpha():
            novi += znak
    return novi
```

Ali je bil zadnji znak črka, preverimo preprosto tako, da pogledamo, kateri je bil zadnji znak, ki smo ga dopisali v niz `novi`. Tako zagotovimo, da v `novi` ne bomo zapisali dveh črk zapored. Še preden preverimo zadnji znak, pa za vsak slučaj preverimo, da novi ni prazen; tedaj gre za prvi znak in tega v vsakem primeru dodamo.

Pogoj `not znak.isalpha() or not novi or not novi[-1].isalpha()` ni preveč pregleden, preveč `not`ov ima. Spomnimo se de Morganovega pravila (ali pa ne in naredimo brez njega, po intuiciji) in napišemo

```
        if not (znak.isalpha() and novi and novi[-1].isalpha()):
            novi += znak
```

Znaka ne smemo dodati, če gre za črko in je niz novi neprazen in je njegov zadnji znak črka; ti pogoji so našteti v oklepaju. Če ni tako (`not` pred oklepajem), ga dodamo.

Nalogo lahko rešimo tudi tako, da do elementov dostopamo z indeksi. Takšno rešitev – napisali bi jo predvsem, tisti, ki znajo programirati v kakem drugem, najbrž neskriptnem jeziku – pokažimo le, da uvidimo, da ni nič preprostejša. Nasprotno.

```
def skritopis(besedilo):
    if not besedilo:
        return ""
    novi = besedilo[0]
    for i in range(1, len(besedilo)):
        if not (besedilo[i].isalpha() and besedilo[i - 1].isalpha()):
            novi += besedilo[i]
    return novi
```

V `novi` dodamo prvi znak besedila (razen, če je niz `besedilo` prazen; v tem primeru takoj vrnemo prazen niz, sicer bi funkcija javila napako v naslednji vrstici, ko bi poskušali priti do prvega znaka praznega niza). Nato gremo prek besedila, začenši z znakom z indeksom 1. Vsak znak dodamo, če ni res, da je tako ta znak kot njegov predhodnik črka.

Kdor se je že seznanil z regularnimi izrazi (ki sicer niso ravno osnovno znanje), je napisal veliko krajšo rešitev.

```
def skritopis(s):
    import re
    return re.sub("\w+", lambda mo: mo.group()[0], s)
```

Za konec rešimo poenostavljeno nalogo, ki predpostavi, da so v nizu le besede, ločene s presledki.

```
def skritopis(besedilo):
    novo = ""
    for beseda in besedilo.split():
        novo += beseda[0] + " "
    return novo[:-1]
```

Pripravimo prazen niz, razbijemo besedilo na besede in prve črke le-teh dodajamo v novi niz. Za vsako črko dodamo še presledek. Na koncu niza ostane en odvečni presledek; rešimo se ga tako, da vrnemo niz brez zadnjega znaka.

Ljubitelji enovrstičnih programov to opravijo z enovrstičnim programom.

```
def skritopis(besedilo):
    return " ".join(beseda[0] for beseda in besedilo.split())
```

177. Igra z besedami

Za začetek predpostavimo, da vemo za `defaultdict`. Navsezadnje smo tudi v nalogi namignili, da utegne priti prav.

```
from collections import defaultdict

def skrij(beseda):
    crk = defaultdict(int)
    for c in beseda:
        crk[c] += 1
    nova_beseda = ""
    for crka, pojavitev in sorted(crk.items()):
        nova_beseda += crka + str(pojavitev)
    return nova_beseda
```

V prvi zanki preštejemo, kolikokrat se pojavi katera črka, v drugi sestavimo novo besedo. Iz slovarja bomo vzeli pare `crk.items()`, ki bodo sestavljeni iz terk (črka, število pojavitev). Uredimo jih; `sorted` bo terke uredil po prvem elementu, črki. Če bi bila prva elementa dveh terk enaka, bi upošteval drugi element, vendar se to v tem primeru ne bo zgodilo. Novo besedo sestavimo tako, da lepimo skupaj črke in število njihovih pojavitev, ki jih spremenimo v niz (`str(pojavitev)`).

`Counter`, na katerega smo prav tako namignili v besedilu naloge, nas reši preštevanja.

```
from collections import Counter

def skrij(beseda):
    crk = Counter(beseda)
    nova_beseda = ""
    for crka in sorted(crk):
        nova_beseda += crka + str(crk[crka])
    return nova_beseda
```

Če se spomnimo metode `join` in znamo delati z generatorji, se znebimo še druge zanke.

```
from collections import Counter

def skrij(beseda):
    return "".join(crka + str(pojavitev)
                   for crka, pojavitev in sorted(Counter(beseda).items()))
```

Če ne znamo, pa nič hudega.

178. Srečanje čebel

Ob čebeli skupaj bosta potrebovali toliko časa, kolikor je cvetov (ker potrebujeta sekundo za vsak cvet) in kolikor je nektarja (ker porabita sekundo za vsako enoto). Obe skupaj bosta torej potrebovali `len(vrt) + sum(vrt)` sekund. Če za eno čebelo ugotovimo, kje bo, ko mine pol tega časa, vemo, kje se bosta srečali.

```
def srecanje(vrt):
    cas = (len(vrt) + sum(vrt)) / 2
    cvet = -1
    while cas > 0:
        cvet += 1
        cas -= 1 + vrt[cvet]
    if cas == 0:
        cvet += 0.5
    return cvet
```

Najprej torej izračunamo skupni čas oziroma polovico le-tega, potem pa simuliramo gibanje čebele. V začetku je pred ničtim cvetom (`cvet = -1`). Dokler se ji čas ne izteče (`while cas > 0`), gre v vsakem koraku na naslednji cvet in porabi (s prehodom in na cvetu) `1 + vrt[cvet]` časa.

Na koncu moramo popaziti na možnost, da je čebela na polovici časa ravno opravila z enim od cvetov. V tem primeru bo odletela naprej - druga pa z druge strani. Srečali se bosta na sredi, zato prištejemo 0.5.

Če ste se z nalogo mučili, ker so bili odgovori stalno za 1 ali za 0.5 napačni, se ne vznemirjajte. S takimi nalogami se mučimo vsi, saj zahtevajo zelo natančno razmišljanje in preštevanje.

179. Odvečni presledki

Rešitev je podobna prejšnji. Gremo prek niza in pregledujemo zaporedne znake. Najprej naivno napačna rešitev.

```
def presledki(s):
    odvecnih = presledkov = 0
    for i in range(len(s) - 1):
        if s[i] == " ":
            presledkov += 1
        if s[i] == " " and s[i + 1] == " ":
            odvecnih += 1
    if presledkov == 0:
        return 0
    return odvecnih / presledkov
```

Tole naj bi štelo, koliko je vseh presledkov in koliko je odvečnih, torej dvojnih. Funkcija ne deluje: pomislimo le, kaj se zgodi, ko naleti na trojni presledek. To bi moralo povečati število pravih, potrebnih presledkov za 1 in število odvečnih prav tako. V resnici pa poveča število potrebnih presledkov za 3, število odvečnih pa za 2, saj najde dva para dvojnih presledkov (prvi in drugi ter drugi in tretji presledek v našem trojnem presledku).

Šteti moramo drugače, šteli bomo zadnje presledke. Potrebnih presledkov med besedami je toliko, kolikor je presledkov, ki jim sledi nekaj, kar ni presledek. Tudi če imamo trojen presledek, bomo šteli le zadnjega, saj le zadnjemu sledi nekaj, kar ni presledek. Podobno bomo počeli z večkratnimi presledki: zanima nas, koliko je parov zaporednih presledkov, ki jim sledi nekaj, kar ni presledek. Tako v trojnem presledku spet gledamo le zadnji par, saj le njemu sledi nepresledek.

```
def presledki(s):
    odvecnih = presledkov = 0
    for i in range(len(s) - 2):
        if s[i] == " " and s[i + 1] != " ":
            presledkov += 1
        if s[i] == " " and s[i + 1] == " " and s[i + 2] != " ":
            odvecnih += 1
    if presledkov == 0:
        return 0
    return odvecnih / presledkov
```

Prvi pogoj preveri, ali gre za zadnji (lahko tudi edini) presledek iz skupine presledkov: i-ti znak je presledek, to kar mu sledi, pa ne. Drugi preveri, ali gre za zadnja presledka iz večkratnega presledka: `i`-ti in `i+1`-vi znak sta presledka, naslednji pa ne. Po koncu zanke vrnemo razmerje med odvečnimi in potrebnimi presledki, predtem pa za vsak slučaj preverimo, ali sploh imamo kak presledek; če ga ni, kar takoj vrnemo 0, da se izognemo napaki zaradi deljenja z 0.

Program ne deluje povsem pravilno: za niz `"a    b c"` bi vrnil 1 in ne 0.5, ker gre zanka le do predpredzadnjega znaka – i ne pride tako daleč, da bi vseboval indeks presledka med b in c. Zanko je bilo potrebno spustiti do `range(len(s)-2)` zato, da smo smeli preverjati `s[i+2]`. Eden možnih popravkov je

```
def presledki(s):
    odvecnih = presledkov = 0
    for i in range(len(s)-1):
        if s[i] == " " and s[i + 1] != " ":
            presledkov += 1
        if i < len(s)-2 and s[i] == " " and s[i + 1] == " " and s[i + 2] != " ":
            odvecnih += 1
    if presledkov == 0:
        return 0
    return odvecnih / presledkov
```

Zanko smo spustili en znak dlje in zato k pogoju za večkratne presledke dodali preverjanje, ali še smemo zahtevati `s[i+2]`. (Razmislimo: če ne velja več `i < len(s)-2` imamo torej `i == len(s)-2`. Tedaj je `i+2` ravno `len(s)`, največji dovoljeni indeks pa je `s[len(s)-1]`.

Druga možnost je, da pustimo zanko takšno, kot je, in primer, ko je predzadnji znak presledek, zadnji pa ne, obravnavamo posebej.

```
def presledki(s):
    odvecnih = presledkov = 0
    for i in range(len(s)-2):
        if s[i] == " " and s[i + 1] != " ":
            presledkov += 1
        if s[i] == " " and s[i + 1] == " " and s[i + 2] != " ":
            odvecnih += 1
    if len(s) > 2 and s[-2] == " " and s[-1] != " ":
        presledkov += 1
    elif presledkov == 0:
        return 0
    return odvecnih / presledkov
```

Pri pisanju pogoja smo bili previdnih: preden preverimo predzadnji in zadnji znak, se prepričamo še, da je niz v resnici dolg vsaj dva znaka.

Stavek `if presledkov == 0` smo spremenili v `elif presledkov == 0`. Če je namreč resničen pogoj pred njim, smo število presledkov povečali za 1, zato ni prav nobene potrebe, da preverjamo, ali smo slučajno brez njih. Program bi enako pravilno deloval tudi, če bi `if` ostal `if`, bil bi le nekoliko počasnejši (kar ni tako pomembno), predvsem pa gre za eleganco.

Gotovo pa mnoge že ves čas branja tišči, da bi namesto

```
    if presledkov == 0:
        return 0
    return odvecnih / presledkov
```

pisal kar

```
    return presledkov and odvecnih / presledkov
```

Če je `presledkov` enak 0, je to neresnično in Python ve, da je vrednost izraza enaka 0, še preden gre računat usodni drugi del izraza, `odvecnih / presledkov`, ki bi se končal z napako zaradi deljenja z 0.

In ko smo ravno pri eleganci: z dopolnjenima rešitvama nismo pretirano zadovoljni. Lepe funkcije so funkcije, ki nimajo posebnih primerov, dodatnih pogojev pred, med ali v zanki. Če

se jim je mogoče izogniti, se jim izognemo. Ne le, da bodo programi krajši; posebni pogoji so lepi viri napak. Razlika med izdelkom dobrega in slabega študenta je pogosto prav v tem, da prvi zna rešiti problem brez sitnih dodatnih pogojev, drugi pa se razpiše.

Kako pa to nalogo reši spretnejši študent? Ker ve za regularne izraze, napiše

```
def presledki(s):
    import re
    return len(re.findall("\s\s+", s)) / (len(re.findall("\s+", s)) or 1)
```

180. Kričiš

Kdor je to nalogo reševal podobno kot prejšnji dve, ima sicer za seboj koristno vajo, tule pa bomo storili preprosteje. Z zanko bomo šli do predzadnje črke, iz niza izrezovali pare črk in se spraševali, ali sta obe črki veliki. Čim naletimo naj kaj takšnega, vrnemo `True`.

```
def kricis(s):
    for i in range(len(s) - 1):
        if s[i:i+2].isalpha() and s[i:i+2].isupper():
            return True
    return False
```

Ker poskušamo bralcu počasi precediti sline po regularnih izrazih, pokažimo še rešitev z njimi.

```
def kricis(s):
    return re.search("[A-Z]{2,}", s) != None
```

181. Napadalne kraljice

Bistvo vaje je v tem, da sestavimo kup drobnih funkcij, ki se kličejo med sabo. Če bralec ni reševal tako, naj ne bere naprej, temveč se ponovno loti reševanja.

Stolpec je prost, če v seznamu koordinat ni nobene, katere prvi znak je enak oznaki stolpca. Spet ista fraza kot praštevila.

```
def stolpec_prost(stolpec, postavitev):
    for p in postavitev:
        if p[0] == stolpec:
            return False
    return True
```

Naslednja - poišči proste stolpce - je res preprosta (no, enako bomo rekli tudi pri vseh naslednjih, kar navadite se). Gremo prek seznama stolpcev in za vsakega pokličemo gornjo funkcijo. Če je prost, ga dodamo na seznam prostih stolpcev.

```
def prosti_stolpci(postavitev):
    prosti = []
    for s in "abcdefgh":
        if stolpec_prost(s, postavitev):
            prosti.append(s)
    return prosti
```

Pa naslednja? Potrebujemo prvi prost stolpec, napisali pa smo si že funkcijo, ki vrne vse proste stolpce? Pokličemo funkcijo, ki vrne vse proste stolpce in če seznam ni prazen, vrnemo prvega. Če je, ne vrnemo ničesar in funkcija s tem vrne `None`, kot vse funkcije, ki ne vračajo ničesar.

```
def prost_stolpec(postavitev):
    prosti = prosti_stolpci(postavitev)
    if prosti:
        return prosti[0]
```

Dve polji se napadata, če sta v istem stolpcu (`polje1[0] == polje2[0]`), v isti vrstici (`polje1[1] == polje2[1]`) ali pa na isti diagonali. Slednje merimo tako, da pretvorimo vse koordinate v njihove kode ASCII. Če je absolutna vrednost razlik koordinat enaka – pomaknemo se enako stolpcev levo/desno kot vrstic gor/dol – sta polji na isti diagonali.

```
def napada(polje1, polje2):
    return polje1[0] == polje2[0] or polje1[1] == polje2[1] or \
           abs(ord(polje1[0]) - ord(polje2[0])) == \
               abs(ord(polje1[1]) - ord(polje2[1]))
```

Dano polje napadajo tiste kraljice, ki stojijo na poljih, ki napadajo dano polje. Napadalke bomo zbirali v istoimenskem seznamu; za vsako kraljico iz dane postavitve preverimo, ali napada podano polje in jo po potrebi dodamo na seznam.

```
def napadajo(polje, postavitev):
    napadalke = []
    for kraljica in postavitev:
        if napada(kraljica, polje):
            napadalke.append(kraljica)
    return napadalke
```

Naslednja funkcija pravi, da je polje napadeno, če ga kaka kraljica napada.

```
def napadeno(polje, postavitev):
    return len(napadajo(polje, postavitev)) > 0
```

Prosta polja so polja, ki niso napadena. Gremo prek vrstic, jih sestavimo s stolpcem (`stolpec + vrstica`) in preverimo, ali je polje napadeno. Če ni, ga dodamo na seznam prostih.

```
def prosto_v_stolpcu(stolpec, postavitev):
    prosta = []
    for vrstica in "12345678":
        if not napadeno(stolpec + vrstica, postavitev):
            prosta.append(stolpec + vrstica)
    return prosta
```

In potem pride prva (in zadnja) resna naloga (recimo). Potrebujemo pare kraljic, torej naredimo dve zanki. Da pa bi vsak par dobili le enkrat, "enumeriramo" kraljice. Spremenljivka

`i` vsebuje indeks prve kraljice; v notranji zanki pogledamo le vse kraljice do `i`-te. Če se tidve polji napadata, dodamo par kraljic v seznam napadajočih se.

```
def napadajoce_se(postavitev):
    napadajoce = []
    for i, kraljica1 in enumerate(postavitev):
        for kraljica2 in postavitev[:i]:
            if napada(kraljica1, kraljica2):
                napadajoce.append((kraljica1, kraljica2))
    return napadajoce
```

In še zadnja: postavitev je legalna, če vsebuje osem kraljic in je seznam napadajočih se prazen.

```
def legalna(postavitev):
    return len(postavitev) == 8 and not napadajoce_se(postavitev)
```

"Industrijske" rešitve teh nalog so krajše. Kdor jih ne razume, naj se vrne k njim kasneje.

```
def stolpec_prost(stolpec, postavitev):
    return not any(x[0] == stolpec for x in postavitev)

def prosti_stolpci(postavitev):
    return [s for s in "abcdefgh" stolpci if stolpec_prost(s, postavitev)]

def prost_stolpec(postavitev):
    prosti = prosti_stolpci(postavitev)
    if prosti:
        return prosti[0]

def napada(polje1, polje2):
    return polje1[0] == polje2[0] or \
           polje1[1] == polje2[1] or \
           abs(ord(polje1[0]) - ord(polje2[0])) ==
               abs(ord(polje1[1]) - ord(polje2[1]))

def napadajo(polje, postavitev):
    return [kraljica for kraljica in postavitev if napada(kraljica, polje)]

def napadeno(polje, postavitev):
    return bool(napadajo(polje, postavitev))

def prosto_v_stolpcu(stolpec, postavitev):
    return [stolpec + vrstica for vrstica in vrstice
            if not napadajo(stolpec + vrstica, postavitev)]

def napadajoce_se(postavitev):
    return [(kraljica1, kraljica2)
            for i, kraljica1 in enumerate(postavitev)
            for kraljica2 in postavitev[:i]
            if napada(kraljica1, kraljica2)]

def legalna(postavitev):
    return len(postavitev) == 8 and not napadajoce_se(postavitev)
```

182. Obljudeni stolpci

Če črko pretvorimo v kodo ASCII in odštejemo 97 (ker ima `"a"` kodo 97, `"b"` 98 in tako naprej), bomo dobili številke med 0 in 7, kar bi lahko bili ravno indeksi seznama, ki ga moramo sestaviti – seznama, v katerem bo vsak element ustrezal enemu stolpcu in bo povedal število figur v njem.

Da rešimo prvi del naloge, torej spišemo funkcijo, ki sestavi seznam osmih ničel, nato pa gre prek seznama obljudenih polj in povečuje ustrezne števce.

```
def po_stolpcih(s):
    po = [0] * 8
    for polje in s:
        po[ord(polje[0]) - ord("a")] += 1
    return po
```

Malenkost zvitejša rešitev razpakira polje v vrstico in stolpec.

```
def po_stolpcih(s):
    po = [0] * 8
    for stolpec, vrstica in s:
        po[ord(stolpec) - ord("a")] += 1
    return po
```

Seznam vsebuje polja v obliki dvoznakovnih nizov, na primer "e8". V zanki to razpakiramo v spremenljivki `stolpec` in `vrstica` – ker sta znaka ravno dva, bo to delovalo. (Spremenljivke vrstica sicer ne potrebujemo, vendar jo navajamo zato, da bo Python razpakiral prvi znak v `stolpec`. Po nekakšnih vodilih naj bi takšnim, neuporabnim spremenljivkam, dali ime _, torej `for stolpec, _ in s:`.)

Med temi stolpci je zdaj potrebno vrniti tistega, ki je najbolj obljuden. Takšne naloge srečujemo stalno: podobne so iskanju največjega elementa seznama, le da si moramo zapomniti tudi njegov indeks.

```
def naj_stolpec(s):
    po = po_stolpcih(s)
    naj_s = naj_v = 0
    for i, v in enumerate(po):
        if v > naj_v:
            naj_s, naj_v = i, v
    if naj_v == 0:
        return None
    return chr(97 + naj_s)
```

183. Banka

Prva funkcija bi morala biti za nas že trivialna: sestavimo prazen seznam, gremo čez seznam transakcij in zlagamo imena v prazni seznam, ki ga na koncu vrnemo kot rezultat.

```
def klienti(transakcije):
    imena = []
    for kdo, transakcija in transakcije:
        if kdo not in imena:
            imena.append(kdo)
    return imena
```

Namesto imena transakcije bi lahko uporabili ime _, ki ga uporabljamo za spremenljivke, ki jih v bistvu ne potrebujemo in jim imamo le zato, ker jih moramo imeti. V gornjem primeru jo potrebujemo le zato, ker gremo prek seznama parov. Zanko bi lahko napisali tudi malo drugače, namreč

```
def klienti(transakcije):
    imena = []
    for e in transakcije:
        if e[0] not in imena:
            imena.append(e[0])
    return imena
```

Vendar je bila prva rešitev, priznajmo, preglednejša.

Izračunati, koliko denarja ima neka oseba, je podobno računanju vsote elementov seznama, le da seštevamo le zneske določene osebe. Preden torej prištejemo znesek transakcije k vsoti, moramo preveriti ime.

```
def bilanca(transakcije, ime):
    skupaj = 0
    for kdo, znesek in transakcije:
        if kdo == ime:
            skupaj += znesek
    return skupaj
```

Ime najbogatejšega klienta lahko poiščemo tako, da združimo obe funkciji: s prvo dobimo imena vseh klientov in z drugo izvemo premoženje posameznega med njimi. Če je le-to večje od premoženja najbogatejšega doslej, si zapomnimo njegovo ime in premoženje.

```
def najbogatejsi(transakcije):
    naj_ime, naj_denarja  = "", 0
    for ime in klienti(transakcije):
        znesek = bilanca(transakcije, ime)
        if znesek > naj_denarja:
            naj_ime, naj_denarja = ime, znesek
    return naj_ime
```

In zdaj, končno, še računovodja. Ker v tem poglavju ne uporabljamo slovarjev – kaj šele slovarjev s privzetimi vrednostmi – se je lotimo brez njih, z golimi seznami. To bo kar nadležno.

Sestavimo prazen seznam, ki ga bomo napolnili s tem, kar mora vračati funkcija. Za vsako transakcijo (zunanja zanka) poiščemo tisti element, ki ustreza imenu osebe, ki je izvedla transakcijo (notranja zanka) in ustrezno spremenimo znesek. Če je ne najdemo (ker je v knjigi še ni), jo dodamo v knjigo.

```
def racunovodja(transakcije):
    knjiga = []
    for kdo, znesek in transakcije:
        for racun in knjiga:
            if racun[0] == kdo:
                racun[1] += znesek
                break
        else:
            knjiga.append([kdo, znesek])
    return knjiga
```

Bodite pozorni, kje je `else`: pripada zanki `for` in ne za stavku `if`. Zgoditi se mora namreč, če še ni zgodil `break`.

Hm. Nismo maloprej smo razkladali o tem, kako je `for ime, transakcija in transakcije` elegantnejše od `for e in transakcije`, saj moramo potem pisati `e[0]` namesto preprostega in jasnega `ime`. Čemu ne tudi tu tako:

```
def racunovodja(transakcije):
    knjiga = []
    for kdo, znesek in transakcije:
        for ime, doslej in knjiga:
            if ime == kdo:
                doslej += znesek
                break
        else:
            knjiga.append([kdo, znesek])
    return knjiga
```

Razlog, da to ne deluje, se skriva v tem, kako delujejo spremenljivke v Pythonu: ko bi spreminjali vrednost `doslej`, bi spreminjali le vrednost `doslej`, ne pa ustrezne vrednosti v seznamu. Spremenljivka `doslej` ni sinonim za ta in ta element seznama `transakcije`.

Oglejmo si, kako bi se to rešilo s slovarji.

```
def racunovodja(transakcije):
    knjiga_s = {}
    for kdo, znesek in transakcije:
        if kdo in knjiga_s:
            knjiga_s[kdo] += znesek
        else:
            knjiga_s[kdo] = znesek
    knjiga = []
    for kdo, koliko in knjiga_s.items():
        knjiga.append([kdo, koliko])
    return knjiga
```

S slovarji s privzetimi vrednostmi gre še lažje.

```
import collections

def racunovodja(transakcije):
    knjiga_s = collections.defaultdict(int)
    for kdo, znesek in transakcije:
        knjiga_s[kdo] += znesek
    knjiga = []
    for kdo, koliko in knjiga_s.items():
        knjiga.append([kdo, koliko])
    return knjiga
```

Zoprnija na koncu, ki je v drugem primeru dolga pol funkcije, je potrebna samo zaradi tega, ker je naloga zahtevala, naj bo rezultat seznam seznamov. Tako pa je zahtevala, ker je hotela biti naloga na temo seznamov, ne slovarjev.

184. Srečni gostje

Naloga ni zahtevna, le precej pogojev je v rešitvi. Motita nas dve stvari.

Prva. Ker je miza okrogla, moramo zadnjega gosta obravnavati posebej. Naj bo `n` število gostov, torej `n = len(razpored)`. Potem bi morali najbrž pisati zanko `for i in range(n-1)` (pazite, `n-1`, gremo torej samo do vključno predzadnjega gosta, tistega z indeksom `n-2`) in potem ugotavljamo srečo `i`-tega gosta tako, da ga primerjamo z gostoma `i-1` in `i+1`. Zadnjega gosta obravnavamo posebej - gosta `n-1` primerjamo z gostom `n-2` in gostom `0`.

Trik, ki ga lahko uporabimo, da se temu izognemo, je preprost: gosta `i` primerjamo z gostoma `i-1` in `(i+1)%n`. Dokler je `i` manjši od `n-1`, je `(i+1)%n` isto kot `i+1`. Pri nerodnem zadnjem gostu, ko je `i` enak `n-1`, pa je `(i+1)%n` enako `n%n` torej `0`. Moduli, ostanki po deljenju, nas vedno rešujejo pri krožnih zadevah – spomnimo se le, koliko gorja so nam prihranili pri nalogi An ban pet podgan! Z ničtim gostom pa sploh ni težav: primerjali ga bomo z gostoma -1 in 1, -1 pa je prav zadnji gost.

Druga. Pogoji so malo zoprni: če je gost ženska, morata biti soseda moška. Če je gost moški, morata biti sosedi ženski. Vendar se da to povedati veliko preprostje: spol gosta mora biti drugačen od spola sosedov. Torej: oseba je srečna, če funkcija `je_zenska`, ki smo jo napisali v davni nalogi Spol iz EMŠO, zanjo vrača drugačno vrednost kot za njena soseda.

```
def je_zenska(emso):
    return emso[9] >= "5"

def stevilo_srecnezev(razpored):
    srecnih = 0
    n = len(razpored)
    for i in range(n):
        if je_zenska(razpored[i-1]) != je_zenska(razpored[i]) !=
je_zenska(razpored[(i + 1) % n]):
            srecnih += 1
    return srecnih
```

C'est tout! Če se trikov ne domislimo, je rešitev nekoliko daljša, a v osnovi nič težja.

185. Gostoljubni gostitelji

Namig je povedal vse: najprej naredimo seznam žensk in seznam moških. Potem ženske in moške izmenično dodajamo v seznam, na koncu pa oddamo še nepoparjene. S kom začnemo, je vseeno; če kdo misli, da se splača začeti s tistimi, ki jih je manj, naj se spomni, da je miza okrogla. (Če še ni jasno, naj vzame v roke starodavno orodje programerjev, papir in svinčnik.)

```
def razporedi(gostje):
    zenske = []
    moski = []
    for gost in gostje:
        if je_zenska(gost):
            zenske.append(gost)
        else:
            moski.append(gost)
    parov = min(len(zenske), len(moski))
    razpored = []
    for i in range(parov):
        razpored.append(moski[i])
        razpored.append(zenske[i])
    razpored += moski[parov:] + zenske[parov:]
    return razpored
```

Gornji program je začetniško okoren, le tik pred koncem, v predzadnji vrstici smo izgubili potrpljenje in uporabili droben trik. Tiste, ki ostanejo, prištejemo tako, da dodamo vse nepoparjene ženske in vse nepoparjene moške, `zenske[parov:]` in `moski[parov:]`. Eden od teh dveh seznamov je vedno prazen (kadar imajo vsi gosti srečo, sta prazna celo oba), vendar nam prištevanja praznih seznamov res nihče ne bo zameril.

Zdaj nalogo rešimo še malo elegantneje.

```
def razporedi(gostje):
    zenske = [gost for gost in gostje if je_zenska(gost)]
    moski = [gost for gost in gostje if not je_zenska(gost)]
    razpored = []
    for par in zip(moski, zenske):
        razpored += par
    parov = len(razpored)//2
    razpored += moski[parov:] + zenske[parov:]
    return razpored
```

Gre seveda tudi krajše, tudi, jasno, v eni vrstici, a tale rešitev je že povsem elegantna in vsaj eno (srečno) študentko (v pomenu te naloge so študentke naše fakultete žal večinoma srečne) smo opazili pisati nekaj v tem slogu, kar, šovinistično gledano, jasno dokazuje, da ta rešitev, čeprav uporablja za nekatere strašljivi `zip`, ni prav nič zahtevnega.

Sicer pa ta naloga nudi še nekaj zabavnih rešitev. Spodnji bi naredili veliko krivico, če je ne bi objavili:

```
def razporedi(gostje):
    zenske = [gost for gost in gostje if je_zenska(gost)]
    moski = [gost for gost in gostje if not je_zenska(gost)]
    razpored = []
    while zenske or moski:
        for x in zenske, moski:
            if x:
                razpored.append(x.pop())
    return razpored
```

Poglobite se vanjo; ko jo boste razumeli, se boste počutili, kot da ste pokapirali dober vic.

Povemo še enega?

```
def razporedi(gostje):
    gostje = sorted(gostje, key=lambda x:x[9])
    razpored = []
    for g1, g2 in zip(gostje, reversed(gostje)):
        razpored += [g1, g2]
    return razpored[:len(gostje)]
```

Tega bomo kar razložili, je čisto poučen. (Mimogrede, v tem duhu napišemo tudi rešitev v eni vrstici, če bi nas ravno srbeli prsti.) Goste uredimo po deveti števki EMŠO. Urejanje bo postavilo moške na začetek, ženske na konec. Nato zadrgnemo skupaj seznam gostov in obrnjeni seznam gostov (`reverse(gostje)` je isto kot `gostje[::-1]`, le da Python v resnici obrača seznam sproti, namesto da bi sestavil nov seznam). Pari `g1`, `g2` tako predstavljajo goste, ki jih jemljemo istočasno z začetka seznama in s konca. Zložimo jih v novi seznam, `razpored`. Žal je na tem razporedu vsak gost dvakrat – enkrat ga je pobral `g1`, drugič, z druge strani, `g2`. Zaskrbi nas tudi, kaj imamo v sredini, saj je od tega, ali je gostov sodo ali liho, odvisno, ali smo enkrat z `g1` in `g2` pobrali enega in istega gosta ali ne. Kje je torej potrebno presekati razpored? Zelo preprosto: v razporedu mora biti toliko gostov, kolikor je povabljenih na gostijo, torej vrnemo `razpored[:len(gostje)]`.

186. Po starosti

Da bo rešitev preglednejša, si pripravimo funkcijo, ki ji damo EMŠO, na primer "2012983505012" in vrne rojstni datum v "japonskem" formatu, 19821220, se pravi najprej leto rojstva s štirimi števkami, nato mesec, nato dan (YYYYMMDD).

```
def datum_iz_emso(emso):
    dat = emso[4:7] + emso[2:4] + emso[:2]
    if dat[0] < "8":
        return "2" + dat
    else:
        return "1" + dat
```

Stavek `if` na koncu služi temu, da pri tistih, katerih trimestna letnica se začne z 0-7, dodamo na začetek dvojko (rojeni so po 2000; skripta bo uporabna vse do daljnjega leta 2799), tistim, ki imajo 8 ali 9, pa enico (18xx, 19xx). Ne da bi si morali očitati packarijo, lahko funkcijo tudi malo stlačimo.

```
def datum_iz_emso(emso):
    return ("2" if emso[4] < "8" else "1") + \
           emso[4:7] + emso[2:4] + emso[:2]
```

Odtod je pot lahka. Seznam predelamo tako, da bo imel namesto parov (ime, EMŠO) pare (datum, ime). Tak seznam lahko uredimo s `sort`, saj bo le-ta urejal najprej po prvem elementu para, datumu (če bosta dve osebi rojeni na isti dan, pa ju uredi po imenu). Nato iz urejenega seznama poberemo imena.

```
def po_starosti(s):
    nov_sez = []
    for ime, emso in s:
        nov_sez.append((datum_iz_emso(emso), ime))
    nov_sez.sort()
    rez = []
    for datum, ime in nov_sez:
        rez.append(ime)
    return rez
```

Če poznamo izpeljevanje seznamov, moremo nalogo rešiti tudi hitreje.

```
def po_starosti(s):
    nov_sez = [(datum_iz_emso(emso), ime) for ime, emso in s]
    nov_sez.sort()
    return [ime for datum, ime in nov_sez]
```

Lahko pa rečemo celo kar

```
def po_starosti(s):
    return [ime for datum, ime in sorted((datum_iz_emso(emso), ime) for ime, emso
in s)]
```

187. Ujeme

Naloga je praktično enaka ujemanju črk, če jo sparimo z nalogo Pokaži črke. Tu pač ne seštevamo enic temveč črke. Predelajmo kar osnovno rešitev prejšnje naloge.

```
def ujeme(b1, b2):
    b = ""
    for i in range(min(len(b1), len(b2))):
        if b1[i] == b2[i]:
            b += b1[i]
        else:
            b += "."
    return b
```

Ali, krajše

```
def ujeme(b1, b2):
    b = ""
    for i in range(min(len(b1), len(b2))):
        b += b1[i] if b1[i] == b2[i] else "."
    return b
```

Program nam še malo poenostavi `zip`.

```
def ujeme2(b1, b2):
    b = ""
    for c1, c2 in zip(b1, b2):
        b += c1 if c1 == c2 else "."
    return b
```

Odtod pa nas loči le še droben korak do preproste rešitve v eni sami vrstici.

```
def ujeme3(b1, b2):
    return "".join(c1 if c1 == c2 else "." for c1, c2 in zip(b1, b2))
```

188. Najboljše prileganje podniza

Ker smo že pri nalogi Ujemanje črk napisali funkcijo, ki pove, v koliko črkah se ujemata podana podniza, bomo tule uporabili kar to funkcijo: podajali ji bomo koščke prvega niza, da jih bo primerjala z drugim.

```
def naj_prileg(s, sub):
    najpos = najuj = -1
    for pos in range(len(s)):
        uj = st_ujemanj(s[pos:], sub)
        if uj > najuj:
            najpos, najuj = pos, uj
    return najpos, najuj, s[najpos:najpos + len(sub)]
```

Rešitev s tem spominja na klasiko Poišči največji element, le da ne iščemo elementa, temveč zamik in ne opazujemo velikosti, temveč ujemanje.

189. Deljenje nizov

Naloga je lahko strašno zoprna ali strašno trivialna, odvisno od tega, ali smo pripravljeni razmisliti, preden se lotimo programiranja.

Če je niz "deljiv" s `k`, bo njegova dolžina večkratnik `k` in če vzamemo prvih `len(s)//k` znakov ter jih pomnožimo s `k`, spet dobimo isti niz. Če vse to drži, torej vrnemo teh prvih `len(s)//k` znakov. Sicer ne naredimo ničesar in funkcije, ki ne vračajo ničesar, vemo, vračajo `None`, kot zahteva naloga.

```
def deli_niz(s, k):
    d = s[:len(s)//k]
    if d * k == s:
        return d
```

190. Bralca bratca

Tale je zoprna, če jo hočemo rešiti res povsem prav - tako, da pravilno deluje tudi na najbolj zoprnih primerih. Problem predstavlja namreč srednja knjiga: eden od bratcev bo prebral več in "vmesno" knjigo naj vzame tisti, ki bo prebral "manj več". Spodnja rešitev deluje tako, da Peter zgrabi naslednjo knjigo samo, če ga branje polovice te knjige ne bo pripeljalo čez polovico vseh knjig. Izkaže se, da to da pravilno rešitev.

```
def razdeli_knjige(debeline):
    pol_vsega = sum(debeline)/2
    petrove = 0
    i = 0
    for i in range(len(debeline)):
        if petrove + debeline[i]/2 > pol_vsega:
            break
        petrove += debeline[i]
    return i, len(debeline)-i
```

Rešitev naredi točno, kar smo opisali zgoraj. V `pol_vsega` zapišemo, koliko strani ima polovica police. Nato gremo v zanki čez knjige in knjigo dodamo med Petrove strani, če bo po polovici te knjige še vedno pod polovico police. Sicer pa prekinemo zanko in vrnemo, kolikor naj bi prebral Peter (`i`) in kolikor Pavel (`len(debeline)-i`).

Čemu pa `i = 0` pred zanko?! Saj zanka vendar ve, da mora začeti šteti z 0, ne?! Poskusite pognati program

```
for i in range(0):
    pass
print(i)
```

Spremenljivka `i` ne obstaja. Ker se zanka ni nikoli izvedla, `i`ju ni priredila nobene vrednosti! Za vsak slučaj, če bi bila Peter in Pavel tako zabita, da bi nas ob prazni polici (`debeline=[]`) spraševala, koliko morata prebrati, smo pred zanko postavili `i` na `0`, da bo `return i, len(debeline) - i` vrnil, kar je treba, namreč `0, 0`.

V izogib tem komedijam bi bilo enako dobro (in morda tudi preglednejše) funkcijo začeti z

```
if not debeline:
    return 0, 0
```

Za Python naravnejša (sicer pa smiselno enaka rešitev) bi bila uporaba `enumerate`.

```
def razdeli_knjige(debeline):
    pol_vsega = sum(debeline)/2
    petrove = 0
    i = 0
    for i, debelina_knjige in enumerate(debeline):
        if petrove + debelina_knjige/2 > pol_vsega:
            break
        petrove += debelina_knjige
    return i, len(debeline)-i
```

Kdor želi sprogramirati pravično delitev, v kateri ne bereta vsak z ene strani, temveč je dovoljena poljubna delitev knjig (in bi Pavel prebral le knjigo z 900 stranmi, Peter pa vse

ostale), je toplo vabljen. Gre za enega klasičnih problemov s področja algoritmov, rešitve boste našli, če se boste ozrli za *problemi polnjenja nahbrtnika*.

191. Turnir Evenzero

Naloga zahteva dvojno zanko. Zunanja pravi: ponavljaj, dokler imaš več kot enega tekmovalca. Notranja pravi: pobiraj pare tekmovalcev – 0 in 1, 2 in 3, 4 in 5 ter tako naprej – za vsak par izračunaj zmagovalca in ga uvrsti v naslednji krog. Zunanja zanka bo vrste `while`, saj se pogoj (po slovensko) glasi *dokler* imaš več kot enega tekmovalca. Notranja je `for`, saj z njo štejemo in že pred zanko dobro vemo, do koliko. (Roko na srce, to vemo tudi za zunanjo zanko: izvedla se bo $\log_2 n$-krat. Vendar nam vseeno bolj diši po `while`.) Tekmovalci bodo v seznamu `tekmovalci`; tekmovalce, ki gredo v naslednji krog, bomo zlagali v seznam `naslednji`.

```
def turnir(tekmovalci):
    while len(tekmovalci) > 1:
        naslednji = []
        for i in range(0, len(tekmovalci), 2):
            if (len(tekmovalci[i]) + len(tekmovalci[i + 1])) % 2 == 0:
                naslednji.append(tekmovalci[i])
            else:
                naslednji.append(tekmovalci[i+1])
        tekmovalci = naslednji
    return tekmovalci[0]
```

Ali je vsota črk v imenih soda ali liha, pogledamo kar tako, da seštejemo obe imeni in preverimo dolžino.

Poglejmo še, kako smo premetavali seznama: na začetku vsakega kroga (takoj znotraj zanke `while`) smo sestavili prazen seznam tekmovalcev v drugem krogu. Vanj smo zlagali zmagovalce iz parov tega kroga. Na koncu zanke smo s `tekmovalci = naslednji` zavrgli stari seznam tekmovalcev in poskrbeli, da se bodo v naslednjem krogu pomerili novi.

Na koncu, ko imamo le še enega tekmovalca, vrnemo prvi element seznama, torej preostalega tekmovalca, zmagovalca.

Naloga ima tudi zelo lepo rešitev, ki je krajša, ima le eno zanko in deluje tudi za turnirje, pri katerih število tekmovalcev ni potenca 2.

```
def turnir(x):
    while len(x) > 1:
        o1, o2 = x.pop(0), x.pop(0)
        x.append(o1 if len(o1 + o2) % 2 == 0 else o2)
    return x[0]
```

S seznama poberemo prva dva tekmovalca (za kar je dobro poznati metodo `pop`). Nato *na konec seznama* potisnemo zmagovalca tega para (`o1` če je vsota ime soda, sicer `o2`). To ponavljamo, dokler ne ostane en sam.

Kdor ne razume, naj si predstavlja, da fizično postavi tekmovalce v vrsto. Pokliče prva dva, enega pošlje v klop za poražence, drugega pa na konec vrste. Čez nekaj časa se bo "sam od sebe" začel drugi krog, ko bodo ponovno na vrsti tisti, ki jih je prej poslal na konec...

192. Najdaljše nepadajoče zaporedje

Naloga je nekoliko podobna nalogi Nepadajoči seznam (očitno!), le da tu štejemo dolžino in vsakič, ko je trenutni element manjši od prejšnjega, ne vrnemo `False`, temveč le začnemo šteti od začetka. Funkcija si mora zapomniti, do kod je največ naštela.

```
def najdaljse_nepadajoce(s):
    dolzina = 1
    najdaljsa = 0
    for i in range(1, len(s)):
        if s[i] >= s[i - 1]:
            dolzina += 1
            if dolzina > najdaljsa:
                najdaljsa = dolzina
        else:
            dolzina = 1
    return najdaljsa
```

Spremenljivka `dolzina` meri dolžino trenutnega podzaporedja. Če je trenutno število večje od prejšnjega, povečamo dolžino za 1. Če je podzaporedje že daljše od najdaljšega doslej najdenega, si to zapomnimo tako, da shranimo trenutno dolžino v `najdaljse`.

Sicer, torej če je trenutno število manjše od prejšnjega, se je začelo novo zaporedje in njegova `dolzina` je 1.

Moramo pogoj `dolzina > najdaljsa` res preverjati ob vsakem podaljšanju zaporedja? Ga ne bi raje le tedaj, ko je zaporedja konec? Se pravi, ko naletimo na manjši element, se vprašamo, ali je bilo zaporedje, ki se je pravkar končalo, najdaljše doslej? Tako pridemo do naslednjega programa, ki pa ima manjšo napako.

```
# Ta funkcija ne deluje povsem pravilno!
def najdaljse_nepadajoce(s):
    dolzina = 1
    najdaljsa = 0
    for i in range(1, len(s)):
        if s[i] >= s[i - 1]:
            dolzina += 1
        else:
            if dolzina > najdaljsa:
                najdaljsa = dolzina
            dolzina = 1
    return najdaljsa
```

Program ne deluje, če se seznam konča z najdaljšim zaporedjem. Tako pri zaporedju `[1, 2, 1, 2, 3, 4]` odgovori 2 namesto 4. To lahko uredimo na več načinov: zadnje zaporedje lahko umetno prekinemo tako, da za zadnji element seznama dodamo "stražarja", element, ki je

manjši od zadnjega. To storimo z `s.append(s[-1] - 1)`. To je grdo, saj nas ni nihče "pooblastil", da spreminjamo seznam `s`. Kaj, če smo ta seznam pripravili nekje v nekem drugem delu programa in si ga ne smemo kar tako popackati? Ena možnost je tale.

```
def najdaljse_nepadajoce(s):
    dolzina = 1
    najdaljsa = 0
    len_s = len(s)
    for i in range(1, len_s + 1):
        if i < len_s and s[i] >= s[i - 1]:
            dolzina += 1
        else:
            if dolzina > najdaljsa:
                najdaljsa = dolzina
            dolzina = 1
    return najdaljsa
```

Zanki smo dodali še en krog, ko je i v resnici že prevelik; v tem krogu le preverimo, ali je zadnje podzaporedje daljše od najdaljšega. Funkcija je neučinkovita (čeprav nas tule to najbrž ne boli preveč), ker v vsakem koraku preverja, ali je `i` že dosegel konec seznama. Mimogrede, dolžino seznama smo shranili v `len_s`, da nam ni potrebno stalno klicati funkcije `len`. Klici funkcij so "dragi": namesto da bi stalno klicali in klicali eno in isto funkcijo, si rezultat raje nekam zabeležimo.

Še najboljša rešitev – in tudi najhitrejša – je na koncu zanke preveriti, ali je zadnje, "neprekinjeno" zaporedje daljše od najdaljšega.

```
def najdaljse_nepadajoce(s):
    if not s:
        return 0
    dolzina = 1
    najdaljsa = 0
    for i in range(1, len(s)):
        if s[i] >= s[i - 1]:
            dolzina += 1
        else:
            if dolzina > najdaljsa:
                najdaljsa = dolzina
            dolzina = 1
    if dolzina > najdaljsa:
        najdaljsa = dolzina
    return najdaljsa
```

Načelno se izogibamo temu, da večkrat ponavljamo enake dele programa. To navadno pomeni, da nečesa nismo dobro razmislili ali da bi morali raje definirati funkcijo. In če v kodi, ki se takole ponovi, najdemo napako, moramo popraviti kodo na več mestih in lahko katero od mest spregledamo. V gornjem programu pa je ponovljene kode tako malo in tako kratka je, da nas ne moti.

Pazite na začetek: za razliko od prejšnje funkcije ta ne bi delovala pravilno pri praznih seznamih. Odgovorila bi, da obstaja podzaporedje dolžine 1. S tem najprejprosteje opravimo na začetku.

Pokažimo še rokohitrsko različico programa. Naj razume, kdor hoče.

```
def najdaljse_nepadajoce(s):
    dolzina = 1
    najdaljsa = 0
    for i in range(1, len(s)):
        najdaljsa = max(najdaljsa, dolzina)
        dolzina = dolzina * (s[i] >= s[i - 1]) + 1
    return min(len(s), najdaljsa)
```

Pokažimo še rešitev po vzorcu, po katerem si zapomnimo prejšnji element – kot smo naredili tudi v nalogi Nepadajoče zaporedje. V to obliko lahko predelamo katerokoli od gornjih različic.

```
def najdaljse_nepadajoce(s):
    if not s:
        return 0
    dolzina = najdaljsa = 0
    prejsnji = s[0]
    for trenutni in s:
        if trenutni >= prejsnji:
            dolzina += 1
            if dolzina > najdaljsa:
                najdaljsa = dolzina
        else:
            dolzina = 1
        prejsnji = trenutni
    return najdaljsa
```

Kot v Nepadajoče zaporedje tudi tu na koncu zanke shranimo trenutni element v spremenljivko `prejsnji`, da nam bo na voljo v naslednjem krogu. In tako kot tam tudi tu takrat, ko se zanka izvede prvič, primerjamo prvi element sam s sabo. Da bi se številke izšle, v začetku postavimo `dolzina` na 0. Ker prvi element ni manjši od sebe, bomo povečali `dolzina` z 0 na 1, kar bo ravno prav.

193. Seznam vsot seznamov

Za vsak (notranji) seznam izračunamo njegovo vsoto in jo dodamo v novi seznam, vsote.

```
def vsota_seznamov(s):
    vsote = []
    for ss in s:
        vsota = 0
        for x in ss:
            vsota += x
        vsote.append(vsota)
    return vsote
```

Če uporabimo vdelano funkcijo `sum`, ki vrne vsoto elementov seznama, se funkcija še precej skrajša.

```
def vsota_seznamov(s):
    vsote = []
    for ss in s:
        vsote.append(sum(ss))
    return vsote
```

Če poleg tega vemo kaj o funkcijskem programiranju, postane reč že otročje kratka.

```
def vsota_seznamov(s):
    return list(map(sum, s))
```

Pri reševanju druge naloge se ne ubijajmo s seštevanjem, temveč že kar takoj uporabimo `sum`.

```
def najvecja_vsota(s):
    najvecji = []
    najvecja_vsota = 0
    for ss in s:
        vsota = sum(ss)
        if vsota > najvecja_vsota:
            najvecji, najvecja_vsota = ss, vsota
    return najvecji
```

Lenuhe naj opozorimo, da rešitev, v kateri se izognemo spremenljivki `vsota`,

```
    for ss in s:
        if najvecja_vsota < sum(s):
            najvecji, najvecja_vsota = ss, sum(s)
```

ni najbolj posrečena, ker dvakrat kliče seštevanje. Predstavljajte si, da ima seznam deset milijonov elementov…

V resnici pa noben izkušen programer v Pythonu ne bi rešil naloge na ta način, temveč bi napisal kar

```
def najvecja_vsota(s):
    return max(s, key=sum)
```

194. Veliko, a ne več kot

Če sledimo namigu, dobimo

```
def naj_pod(s, n):
    naj = naj_zac = naj_kon = 0
    for zac in range(len(s)):
        for kon in range(zac, len(s) + 1):
            vs = sum(s[zac:kon])
            if naj < vs <= n:
                naj, naj_zac, naj_kon = vs, zac, kon
    return s[naj_zac:naj_kon]
```

Naloga je za mnoge težka zaradi gnezdenih zank,

```
    for zac in range(len(s)):
        for kon in range(zac, len(s) + 1):
```

Dodatna težava je gornja meja notranje zanke: `kon` moramo spustiti do `len(s) + 1`. Če imamo, recimo, 10 elementov, bo `len(s) + 1` enak `11`, `kon` bo šel do (vključno) `10` in preverjali bomo podseznam `zac:10`, torej do zadnjega elementa. Če bi spustili `kon` le prek `range(len(s))`, bi prišel le do (vključno) `9`, preverjali bi seznam `zac:9` in s tem izpustili zadnji element, element z indeksom `9`.

Znotraj zanke ni nič posebnega. Če je vsota v podseznamu `s[zac:kon]` večja od vseh doslej, a hkratni dovolj majhna, si zapomnimo vsoto, začetek in konec. Če se ne bi spomnili na funkcijo sum, bi dobili, zabavno, še eno zanko znotraj dvojne zanke. Program s trojno zanko!

Na koncu vrnemo najboljše, kar smo našli – podseznam od `naj_zac` do `naj_kon`.

Znamo napisati kaj boljšega? Seveda, vedno.

```
def naj_pod(s, n):
    naj = naj_zac = naj_kon = 0
    for zac in range(len(s)):
        for kon in range(zac, len(s) + 1):
            vs = sum(s[zac:kon])
            if vs > n:
                break
            if naj < vs:
                naj, naj_zac, naj_kon = vs, zac, kon
    return s[naj_zac:naj_kon]
```

Tale popravek je malenkosten, a če bi šlo zares – recimo, da bi program preiskoval kake ogromne sezname – bi se še kako poznal: če je vsota večja od `n`, lahko notranjo zanko prekinemo, saj vemo, da se bo, ko `kon` teče naprej, če še povečevala.

V naslednjem koraku se znebimo klica funkcije `sum` in vsoto računajmo kar sami, mimogrede.

```
def naj_pod(s, n):
    naj = naj_zac = naj_kon = 0
    for zac in range(len(s)):
        vs = 0
        for kon in range(zac, len(s)):
            vs += s[kon]
            if vs > n:
                break
            if naj < vs:
                naj, naj_zac, naj_kon = vs, zac, kon + 1
    return s[naj_zac:naj_kon]
```

Znotraj notranje zanke k vsoti prištejemo vrednost elementa `s[kon]`. Vsoto nastavimo na 0 pred notranjo zanko in znotraj notranje zanke bo vedno vsebovala vsoto od elementov `zac` do (vključno) `kon`. Ne spreglejte, da smo spremenili meje notranje zanke, ki po novem teče samo prek `range(zac, len(s))`, obenem pa smo poskrbeli, da najboljši zgornji meji, `naj_kon`, priredimo `kon + 1`.

Rešitev ima še vedno dvojno zanko. In na prvi pogled je neizogibna: preskusiti moramo vse možne začetke in vse možne konce, ne? Niti ne.

Nalogo bomo rešili tako, da bomo približno istočasno spreminjali `zac` in `kon`. Kadar bo vsota manjša od dovoljene, jo povečamo tako, da premaknemo v desno `kon`. Kadar bo večja od dovoljene, jo zmanjšamo tako, da v desno premaknemo `zac`.

Predstavljajmo si `zac` in `kon`, ko sta nekje sredi seznama – kjerkoli pač že. Možno je dvoje: vsota podseznama med začetkom in koncem je večja od meje ali pa manjša ali enaka meji. Če je večja, je stvar jasna: začetek je potrebno premakniti v desno. Ker po novem vsoto seštevamo sami, ob premiku začetka v desno zmanjšamo vsoto za `s[zac]`. Če je vsota manjša ali enaka meji, preverimo, ali je najboljša doslej. Če je, si zapomnimo vsoto in meji. Če je `kon` prišel že do konca seznama, končamo delo: vsota se ne bo več povečevala, saj lahko premikamo le še `zac`, to pa vsoto zmanjšuje.

```
def naj_pod(s, n):
    naj = naj_zac = naj_kon = zac = kon = vs = 0
    while zac < len(s):
        if vs > n:
            vs -= s[zac]
            zac += 1
        else:
            if vs > naj:
                naj, naj_zac, naj_kon = vs, zac, kon
            if kon == len(s):
                break
            vs += s[kon]
            kon += 1
    return s[naj_zac:naj_kon]
```

Tale rešitev nikakor ni najpreprostejša ali najkrajša, je pa najhitrejša. Zahteva pa veliko pazljivosti pri mejah. Tu smo jo obrnili tako, da `vs` vsebuje vsoto podseznama `s[zac:kon]`, torej z `zac`, a brez `kon`. Tako je bilo najelegantneje.

Od programerja-začetnika takšne rešitve seveda ne pričakujemo. Tole je bolj v domeni predmetov s področja algoritmov in podatkovnih struktur.

195. Nepadajoči podseznami

Rešitev je predvsem vaja iz pozornosti pri delu z indeksi. Videli bomo: če stvar pravilno zastavimo, se vse lepo izide. Ko ste nalogo reševali sami, pa ste se kaj verjetno zaplezali v kakih +1 in -1, ki ste jim morali dodajati ob indeksih, da elementi niso izginjali in se niso podvajali.

```
def nepadajoci(xs):
    res = []
    zac = 0
    for kon in range(1, len(xs)):
        if xs[kon] < xs[kon - 1]:
            res.append(xs[zac:kon])
            zac = kon
    res.append(xs[zac:])
    return res
```

Spremenljivka zac bo vsebovala začetek podseznama. V začetku bo enaka 0. V zanki povečujemo spremenljivko kon, ki predstavlja konec podseznama, dokler ne opazimo, da je kon-ti element manjši od svojega predhodnika. V tem primeru dodamo zadnji nepadajoči seznam – od zac do kon in nastavimo začetek novega podseznama na konec tega, zac = kon, tako da bomo naslednjič kopirali od tega mesta.

Pri tem premetavanju indeksov moramo biti pozorni in natančni: paziti moramo, da ne bi preskočili mejnega elementa ali pa ga prepisali dvakrat. Spomnimo se, kako deluje rezanje: ko rečemo xs[zac:kon], to vključuje zac-ti element, ne pa tudi kon-tega. To je prav: kon-ti element ne pripada več temu podseznamu, saj je manjši od predhodnika. Torej je prav, da ga ne prepišemo v novi podseznam. Pač pa ga ne bomo pozabili v naslednjem krogu: postavili smo zac = kon in ker xs[zac:kon] vključuje začetni element, bo tedaj, v naslednjem krogu, prišel na vrsto tudi element, ki je v tem krogu zadnji.

Ko se zanka konča, prepišemo še zadnje zaporedje, namreč tisto, ki se je končalo, ker je bilo konec seznama, ne pa, ker bi naleteli na manjši element. Premislimo še, ali je tu vse v redu: je lahko ta, zadnji podseznam prazen? Ne. V primeru iz naloge bo zac vseboval indeks dvojke na predpredzadnjem mestu v seznamu, kon pa bo imel indeks šestice na koncu. Ker šestica ni manjša od petice, ki je pred njo, if tega delčka ne bo prepisal. Bi enako dobro delovalo tudi, če bi bil na koncu seznam dolžine 1, recimo, če bi šestici sledila še ena trojka? Da: ko bi naleteli na trojko na koncu, bi (še znotraj zanke, v ifu) prepisali podseznam [2, 5, 6], zac in kon pa bi vsebovala indeks trojke, zadnjega elementa. Zanka bi se iztekla, v res.append(xs[zac:]) pa bi v res dodali še seznam [3].

196. Sodi vs. lihi

Dolgočasna in dolga rešitev: preštejemo, koliko je lihih. Če jih je več kot pol, sestavimo seznam, v katerega naberemo vsa liha števila, sicer seznam, v katerega naberemo vsa soda.

```
def sodi_vs_lihi(s):
    lihih = 0
    for e in s:
        if e % 2 == 1:
            lihih += 1
    t = []
    if lihih > len(s)/2:
        for e in s:
            if e % 2 == 1:
                t.append(e)
    else:
        for e in s:
            if e % 2 == 0:
                t.append(e)
    return t
```

Koliko je lihih, lahko preštejemo tudi malenkost elegantneje: seštejemo ostanke vseh števil po deljenju z 2:

```
    lihih = 0
    for e in s:
        lihih += e % 2
```

Bolj praktično je sestaviti oba seznama in vrniti tistega, ki je daljši. Da se ne bo preveč vleklo, ju sestavimo s pomočjo izpeljanih seznamov.

```
def sodi_vs_lihi(s):
    lihi = [e for e in s if e % 2 == 1]
    sodi = [e for e in s if e % 2 == 0]
    return lihi if len(lihi) > len(sodi) else sodi
```

Rokohitreci pa združijo ideji zadnjih dveh koščkov.

```
def sodi_vs_lihi(s):
    t = sum(e % 2 for e in s) > len(s) / 2
    return [e for e in s if e % 2 == t]
```

Najprej odkrijejo, ali je več sodih ali lihih. Funkcija `sum` bo seštela ostanke po deljenju z 2. Če bo to več kot pol seznama, bo `t` enak `True`, kar je isto kot `1`. Če je več sodih, bo `t` enak `False`, kar je `0`. V drugi vrstici poberemo v nov seznam tista števila, katerih ostanek po deljenju z 2 je enak `t`, torej liha ali soda števila.

197. Gnezdeni oklepaji

Rešitve ni težko sprogramirati, le razmisliti jo je potrebno.

Pregledati je potrebno niz, znak za znakom, in sproti beležiti število "odprtih" oklepajev. Če to pade pod 0, niz ni pravilen. Niz je pravilen tudi, če je število odprtih oklepajev po koncu zanke enako 0, oziroma, obratno: pravilen je, če je število odprtih oklepajev na koncu enako 0.

```
def pravilni_oklepaji(s):
    oklepajev = 0
    for c in s:
        if c == "(":
            oklepajev += 1
        else:
            oklepajev -= 1
            if oklepajev < 0:
                return False
    return oklepajev == 0
```

Rešitev zaresne naloge pa je bistveno bolj zapletena.

```
def pravilni_oklepaji(s):
    oklepajev = 0
    for c in s:
        if c == "(":
            oklepajev += 1
        elif c == ")":
            oklepajev -= 1
            if oklepajev < 0:
                return False
    return oklepajev == 0
```

198. Črkovni oklepaji

Kot smo obljubili, je rešitev kratka, vendar le če poznamo sklad. Če rešitev bere kdo, ki ga ne, mu povejmo: sklad je kup krožnikov. Nanj odlagamo krožnike (`push`, oz. `append`, če uporabljamo Pythonove sezname) in jih z njega jemljemo (`pop`). Krožnike pobiramo v obratnem vrstnem redu, kot smo jih odlagali: najprej vzamemo tistega, ki smo ga odložili zadnjega, saj je na vrhu.

```
def crkovni_oklepaji(s):
    sklad = []
    for c in s:
        if c.isupper():
            sklad.append(c)
        elif not sklad or sklad.pop().lower() != c:
            return False
    return not sklad
```

Ko pridemo do velike črke, jo porinemo na sklad. Če črka ni velika, pa je pač mala in bi morala zaključevati veliko. In tu se lahko zgodita dve napačni stvari. Sklad je lahko prazen; v tem primeru mala črka ne zaključuje ničesar. Če sklad ni prazen, pa se utegne zgoditi, da zadnja črka na njem (ko jo spremenimo v malo) ni enaka tej, do katere smo prišli. Tudi to je narobe; v obeh primerih vrnemo `False`.

Ko je niza konec, mora biti sklad prazen, sicer imamo na njem nezaključene črke. Vrnemo torej `not sklad`, kar bo `True`, če je prazen in `False`, če ni.

Če bo komu pomagalo razumevati: če na konec zanke dodamo

```
        print("Crka: ", c, " --> Sklad: ", sklad)
```

klic

```
crkovni_oklepaji("AaBbACBbDdcDda")
```

izpiše

```
Crka:  A  --> Sklad:  ['A']
Crka:  a  --> Sklad:  []
Crka:  B  --> Sklad:  ['B']
Crka:  b  --> Sklad:  []
Crka:  A  --> Sklad:  ['A']
Crka:  C  --> Sklad:  ['A', 'C']
Crka:  B  --> Sklad:  ['A', 'C', 'B']
Crka:  b  --> Sklad:  ['A', 'C']
Crka:  D  --> Sklad:  ['A', 'C', 'D']
Crka:  d  --> Sklad:  ['A', 'C']
Crka:  c  --> Sklad:  ['A']
Crka:  D  --> Sklad:  ['A', 'D']
Crka:  d  --> Sklad:  ['A']
Crka:  a  --> Sklad:  []
```

Na koncu je sklad prazen, torej smo zmagali.

199. Brez oklepajev

Funkcija bo brala niz in ga, črko za črko, prepisovala v novi niz. Kadar bo naletela na oklepaj, pa si bo zapomnila, da smo znotraj oklepajev in prepisovanje izključila, dokler ne pride do zaklepaja. Čeprav nismo ravno pri prvi nalogi v zbirki, začnimo z začetniško nazornostjo in najprej le prepišimo niz na karseda počasen način.

```
def oklepaji(s):
    nov = ""
    for c in s:
        nov += c
    return nov
```

Zdaj pa dodajmo, kar hoče naloga.

```
def oklepaji(s):
    nov = ""
    v_oklepajih = False
    for c in s:
        if c == "(":
            v_oklepajih = True
        if not v_oklepajih:
            nov += c
        if c == ")":
            v_oklepajih = False
    return nov
```

Kot z vso jasnostjo pove njeno ime, spremenljivka `v_oklepaju` pove, ali se nahajamo znotraj oklepajev. V začetku je `False`; ko naletimo na oklepaj, jo postavimo na `True` in ko na zaklepaj, nazaj na `False`. Pred vrstico, s katero prepisujemo črke, dodamo pogoj: črko prepišemo le, če nismo znotraj oklepajev.

Da preverjamo oklepaj pred prepisovanjem in zaklepaj za njim, jasno kaže na našo veliko zvitost: na ta način dosežemo, da v novi niz ne prepišemo ne oklepajev ne zaklepajev. Če

naletimo na oklepaj, takoj ustavimo prepisovanje – in *že* oklepaj se ne prepiše. Če naletimo na zaklepaj, pa vključimo prepisovanje šele na koncu, tako da se zaklepaj *še* ne prepiše.

Kaj bi tako napisana funkcija vrnila za stavek `"Znaš napisati tudi funkcijo, ki dovoljuje tudi gnezdene oklepaje (kot jih imamo (recimo tule) v tem stavku)?"`? Poskusi: vrne `"Znaš napisati tudi funkcijo, ki dovoljuje tudi gnezdene oklepaje  v tem stavku)?"` Nesreča je v tem, da že ob prvem zaklepaju predpostavi, da smo izven oklepajev. Da bi rešili ta problem, se spomnimo predprejšnje naloge. Funkcija, ki smo jo napisali tam, preverja, ali so oklepaji pravilno gnezdeni. Tu naloga pravi, da smemo predpostaviti, da so, a funkcija nam daje še nekaj: v vsakem trenutku pove, ali se nahajamo znotraj oklepajev ali ne. Sparimo jo s prepisovanjem!

```
def oklepaji(s):
    nov = ""
    oklepajev = 0
    for c in s:
        if c == "(":
            oklepajev += 1
        if oklepajev == 0:
            nov += c
        if c == ")":
            oklepajev -= 1
    return nov
```

Namesto booleove spremenljivke `v_oklepajih`, ki pove le, ali smo ali nismo v oklepaju, imamo `oklepajev`, ki šteje, *kolikokrat* smo v oklepaju. Če nismo (`oklepajev == 0`) prepisujemo, sicer ne.

200. Vsota kvadratov palindromnih števil

Seštevanje kvadratov je preprosto, to bomo znali. Kako pa ugotoviti, ali je število palindrom? Število je palindrom, če ostane po tem, ko ga obrnemo, tako kot prej. Obračanju števil je posvečena posebna naloga, tule se delajmo, da vemo: število `i` je palindrom, če je `str(i)==str(i)[::-1]`. Zdaj imamo vse, kar potrebujemo.

```
vsota = 0
for i in range(1000):
    if str(i) == str(i)[::-1]:
        vsota += i ** 2
print(vsota)
```

Bolj izkušenemu takšno dolgovezenje ne pride na misel in raje napiše

```
print(sum(i ** 2 for i in range(1000) if str(i)==str(i)[::-1]))
```

Zdaj pa rešimo nalogo še tako, da pri tem pokažemo nekaj pameti. Palindromi so

- vsa enomestna števila, torej `i`, kjer je `i` med 1 in 9,

- dvomestna števila, katerih števki (desetice in enice) sta enaki, torej `11 * i`, kjer je `i` med 1 in 9,
- trimestna števila, kateri stotice in enice so enake, torej `101 * i+10 * j`, kjer je `i` med 1 in 9, `j` pa med 0 in 9.

Tako lahko napišemo program, ki bo sestavljal palindrome, namesto da bi preskušal vsako število po vrsti, ali je palindrom ali ne. Takole bi sešteli kvadrate vseh palindromov.

```
vsota = 0
for i in range(1, 10):
    vsota += i ** 2
    vsota += (11 * i) ** 2
    for j in range(10):
        vsota += (101 * i + 10 * j) ** 2
print(vsota)
```

Matematiki bi vedeli tu dodati še kaj in stresti vsoto iz rokava, brez programiranja. Mi je ne bomo.

201. Skupinjenje

Funkcija najprej pripravi prazen seznam, nato pa gre čez stari seznam in vztrajno dodaja njegove elemente v zadnji podseznam novega seznama, `novi[-1].append(x)`. Včasih pa mora pred tem začeti novi podseznam – takrat pač, kadar novi element ni tak kot `zadnji`. V tem primeru torej dodamo nov podseznam, `novi.append([])`, in si zapomnimo, kakšen je (novi) `zadnji` dodani element.

```
def group(xs):
    novi = []
    zadnji = None
    for x in xs:
        if x != zadnji:
            novi.append([])
            zadnji = x
        novi[-1].append(x)
    return novi
```

Funkcija ne deluje pravilno, če se podani seznam začne z elementom `None`. Popravimo ga.

```
def group(xs):
    novi = []
    for x in xs:
        if not novi or x != zadnji:
            novi.append([])
            zadnji = x
        novi[-1].append(x)
    return novi
```

Če gremo v to smer, spremenljivke `zadnji` niti ne potrebujemo: `novi[-1][-1]` je zadnji element zadnjega podseznama, torej zadnji dodani element. Primerjati se moramo z njim.

```
def group(xs):
    novi = []
    for x in xs:
        if not novi or x != novi[-1][-1]:
            novi.append([])
        novi[-1].append(x)
    return novi
```

Zaradi igre oglatih oklepajev, ki enkrat uokvirjajo indekse, drugič sezname, je privlačna tudi tale rešitev.

```
def group(xs):
    novi = []
    for x in xs:
        if not novi or x != novi[-1][-1]:
            novi.append([x])
        else:
            novi[-1].append(x)
    return novi
```

202. Trdnjava

Da rešimo nalogo, moramo znati napisati zanko (to znamo) in jo na primernem mestu prekiniti.

```
def trdnjava(premiki):
    x = y = 0
    for kam, koliko in premiki:
        koliko = int(koliko)
        if kam == "U":
            y -= koliko
        elif kam == "D":
            y += koliko
        elif kam == "L":
            x -= koliko
        elif kam == "R":
            x += koliko
        if x == y == 0:
            return True
    return False
```

Niz, ki podaja premik, razpakiramo, kot bi razpakirali terko. Tule ga razpakiramo v `kam` in `koliko`. To storimo celo kar v zanki – celoten seznam premiki gledamo, kot da bi bil seznam parov in pišemo `for kam, koliko in premiki`, čeprav je v resnici seznam nizov z dvema znakoma. Spremenljivko koliko takoj pretvorimo iz števila v niz, enkrat za vselej, namesto da bi morali pisati `int(koliko)` znotraj vsakega premika.

Če bi bili začetniki, bi morali opozoriti še na to, da `return False` ni znotraj zanke, temveč za njo. A ker imamo že nekaj kilometrine, na ta trik menda ne padamo več.

Špageti iz if-ov nam gredo na živce.[1] Lahko se jih znebimo s prikladnim slovarjem:

```
def trdnjava(premiki):
    x = y = 0
    odmiki = {"U": (0, -1), "D": (0, 1), "L": (-1, 0), "R": (1, 0)}
    for kam, koliko in premiki:
        koliko = int(koliko)
        dx, dy = odmiki[kam]
        x += dx * koliko
        y += dy * koliko
        if x == y == 0:
            return True
    return False
```

Če je, recimo, `kam` enak `"U"`, bosta `dx` in `dy` enaka `0` in `-1`; `dx * koliko` in `dy * koliko` bosta `0` in `-koliko`, torej bomo `x` in `y` spremenili natančno tako, kot je treba.

Še nekoliko krajša rešitev je

```
def trdnjava(premiki):
    x = y = 0
    for kam, koliko in premiki:
        odmiki = {"U": (0, -koliko), "D": (0, koliko),
                  "L": (-koliko, 0), "R": (koliko, 0)}
        koliko = int(koliko)
        dx, dy = odmiki[kam]
        x += dx
        y += dy
        if x == y == 0:
            return True
    return False
```

Je pa, roko na srce, nekoliko počasnejša, saj Python znova in znova sestavlja slovar premikov.

Kot zanimivost pokažimo še rešitev s kompleksnimi števili. Imaginarno enoto, *i*, v večini programskih jezikov, ki poznajo kompleksna števila, pišemo z `j`. Rešitev je tedaj takšna::

```
def trdnjava(premiki):
    smeri = {"U": -1j, "D": 1j, "L": -1, "R": 1}
    p = 0
    for kam, koliko in premiki:
        p += int(koliko) * smeri[kam]
        if p == 0:
            return True
    return False
```

Šahovnico si predstavljamo kot kompleksno ravnino in spremenljivka `p` je kompleksno število; če je trdnjava na polju (4, 2), je `p` enak `4 + 2j`. Prav tako so premiki podani kar s kompleksnimi števili (premik za eno polje dol zapišemo kot `1j`) in zloženi v prikladen slovar.

---

[1] Frajerji znajo tule postokati, da je to zato, ker Python nima stavka `switch`. Tistim, ki ne vedo, kaj je, ga ne bomo razlagali, tistim, ki vedo, pa odgovorimo le, da s tem, ko bi zamenjali `elif` s `case`, pobrisali vse `kam ==` in dodali štiri ukaze `break` ne bi bistveno spremenili programa.

Kako je obrnjen koordinatni sistem – ob premiku navzgor prištevamo ali odštevamo? – je za potrebe te naloge nepomembno.

203. Vsote peterk

Nalogo je preprosto rešiti slabo.

Če ima seznam, recimo, 9 elementov, lahko seštejemo elemente 0:5, 1:6, 2:7, 3:8 in 4:9. Prvi indeks gre torej do dolžine seznama `-5`, drugi do prvega `+5`. Vrnemo največjo vsoto. To bomo že znali, ne?

```
def vsota_peterk(s):
    najvecja = 0
    for zacetek in range(len(s) - 5):
        vsota = sum(s[zacetek:zacetek + 5])
        if vsota > najvecja:
            najvecja = vsota
    return najvecja
```

Problem te rešitve je, da je videti poceni, pa ni. Če iščemo peterke, ni težav. Če bi morali poiskati vsoto desettisočerk, pa bi moral `sum` sešteti 10000 števil. In to bi moral storiti skoraj milijonkrat. Torej bi vsega skupaj seštel deset milijard števil.

Boljše bi bilo računati tekoče vsote. Napišimo program, pa bomo potem razmišljali ob njem.

```
def vsota_nterk(s, n):
    najvecja = vsota = sum(s[:n])
    for i in range(n, len(s)):
        vsota += s[i] - s[i - n]
        if vsota > najvecja:
            najvecja = vsota
    return najvecja
```

Tej funkciji poleg seznama podamo še dolžino podseznama, ki ga želimo seštevati; v prvotni nalogi bi bil `n` enak 5, zdaj je lahko tudi 10000. Najprej seštejemo vsa števila od 0-tega do `n`-tega (z 0 in brez `n`). To bo naša prva in, za začetek, tudi največja `vsota`. Nato pomikamo podseznam naprej: da bi namesto vsote števil `0:n` imeli vsoto števila `1:n+1`, moramo prišteti element `n` in odšteti element `0`. Če je `vsota` večja kot `najvecja`, si jo zapomnimo. V naslednjem krogu bomo prišteli element n+1 in odšteli element 1 ... in tako naprej do konca. Tak program bi na podatkih, s katerimi izziva naloga, opravil le milijon seštevanj in slab milijon odštevanj.

204. Legalni konj

Da bo rešitev elegantnejša, si najprej pripravimo funkciji, ki pretvarjajo iz šahovskih koordinat v kartezične (1 – 8), in funkcijo, ki pove, ali so neke koordinate legalne (obe koordinati morata biti med vključno 1 in 8).

```
def polje_v_koord(polje):
    return ord(polje[0]) - 64, int(polje[1])

def koord_v_polje(x, y):
    return  chr(x + 64) + str(y)

def legalne_koord(x, y):
    return 0 < x < 9 and 0 < y < 9
```

Zdaj gremo lahko prek vseh možnih premikov – (dve levo, ena dol), (dve levo ena gor), (ena levo, dve dol) ... in tako naprej. Vsak premik prištejemo trenutnim koordinatam in preverimo, ali so dobljene koordinate legalne. Če so, jih dodamo v seznam.

```
def moznosti(polje):
    m = []
    x, y = polje_v_koord(polje)
    for dx, dy in ((-2, -1), (-2, 1), (-1, -2), (-1, 2),
                   (1, -2), (1, 2), (2, -1), (2, 1)):
        x1, y1 = x + dx, y + dy
        if legalne_koord(x1, y1):
            m.append(koord_v_polje(x1, y1))
```

Seveda bi šlo tudi brez dodatnih funkcij, celo krajše bi bilo. A takole je preglednejše.

Če se komu zdi spisek vseh možnih potez neeleganten, se mu lahko izogne.

```
for dx in range(-2, 3):
    for dy in range(-2, 3):
        if abs(dx) + abs(dy) == 3:
            x1, y1 = x + dx, y + dy
            ...
```

Koordinata x se spremeni za -2, -1, 1 ali 2, pa še 0 dodamo brez posebne škode. Enako velja za y. Legalne poteze so natančno tiste, pri katerih je vsota absolutnih sprememb enaka 3. Enako dobro bi delovalo tudi

```
        if abs(dx * dy) == 2:
```

205. Skoki

Naloga preskuša, ali znamo napisati zanko in jo po potrebi prekiniti. Očitno zahteva samo, da si zapomnimo trenutno pozicijo (p) in jo v zanki spreminjamo. Šteti moramo poteze in če trenutno pozicija kdaj spet postane 1, vrnemo število potez. Če je trenutna pozicija izven seznama, vrnemo -1. Kako pa zaznamo cikel? O tem govori namig: naredili bomo samo toliko potez, kolikor je dolg seznam. Če v tem času ne pridemo na začetno polje, ne bomo nikoli.

```
def skoki(s):
    p = 0
    n = len(s)
    for i in range(n):
        if s[p] >= n:
            return -1
        p = s[p]
        if p == 0:
            return i + 1
    return -2
```

206. Bobri plešejo

Najprej poskusimo z zanko while, ki zveni nekako naravnejše – `for` uporabim, ko grem prek česa (seznama, datoteke, intervala...), `while` pa, ko zanka teče, dokler ni izpolnjen določen pogoj. Ta recimo, se bo vrtela, dokler ne pridemo na začetno polje.

```
def do_doma(ples, zacetek):
    polje = ples[zacetek]
    korakov = 1
    while polje != zacetek:
        polje = ples[polje]
        korakov += 1
    return korakov
```

Da bi to delovalo, moramo prvi korak narediti že pred zanko; če bi pisali preprosto `polje = zacetek`, bi bil pogoj izpolnjen, še preden zanka prvič steče. Ker že pred zanko naredimo prvi korak, postavimo `korakov` v začetku na `1`, ne na `0`.

Po drugi strani nas vedno, kadar v zanki nekaj štejemo, zamika, da bi morda vendarle uporabili `for`. V resnici nas `for` tudi tokrat ne razočara, če si privoščimo malo drznosti.

```
def do_doma(ples, zacetek):
    polje = zacetek
    for korakov in range(len(ples)):
        polje = ples[polje]
        if pos == zacetek:
            return korakov + 1
```

Zdaj lahko na začetku rečemo `polje = zacetek`; prvi korak bomo v resnici naredili v zanki. Zanko spustimo do toliko korakov, kolikor je dolgo polje ples. Noben ples ne more trajati dlje, razen če se zacikla in to tako, da cikel ne vsebuje začetnega polja (to pa je nemogoče; o tem se tako prepričamo, če poskusimo narisati takšen ples).

Spremenljivko `korak` bo nastavila in povečevala kar zanka. V njej moramo le še narediti korak (`polje = ples[polje]`) in preveriti, ali smo prišli spet na začetek. Če smo, vrnemo `korakov + 1`. Čemu `+1`? Spremenljivka `korakov` bo vedno za 1 premajhna - ko že naredimo

prvi korak, bo imela še vrednost 0; ko naredimo že dva, bo imela vrednost 1..

Posebej bodimo pozorni na to, kje je v gornjem programu `return`: znotraj zanke, vendar v `if`-u. To smo sicer že videli, vendar smo imeli v teh primerih vedno še en return po zanki. Tu ni potreben, saj *vemo*, da se bo v največ `len(ples)` korakih polje ponovilo in bomo prišli na `return`, ki je v zanki. Ta zanka se nikoli ne izteče po naravni poti, temveč jo vedno prej ko slej prekine `return`.

207. Bobri vsi domov

Naloge se lahko lotimo na vsaj dva bistveno različna načina. Lahko simuliramo ples. Predstavljamo si, da so plesalci oštevilčeni. Sestavimo seznam, ki za vsako polje pove, katero številko ima plesalec na njem. Nato "izvajamo" ples, tako da v vsakem koraku zanke prestavimo vse plesalce. Po vsakem koraku preverimo, ali stojijo vsi plesalci na začetnih poljih.

```
def vsi_doma(ples):
    plesalcev = len(ples)
    polja = list(range(plesalcev))
    korakov = 0
    while True:
        novi = [0] * plesalcev
        for i, p in enumerate(polja):
            novi[ples[i]] = p
        korakov += 1
        if novi == list(range(plesalcev)):
            return korakov
        polja = novi
```

V `polja` shrani pozicije vseh plesalcev - ničti element hrani trenutno pozicijo ničtega plesalca in tako naprej, prvi element pozicijo drugega in tako naprej. Potem v vsakem koraku v podoben seznam `novi` izračuna pozicije plesalcev v naslednjem koraku. Preveri, ali stojijo tako kot na začetku (ničti na ničtem, prvi na prvem) in v tem primeru vrne število korakov. Sicer reče prepiše "nova" polja v "trenutna" in ponovi vse skupaj.

Simulacija plesa je odlična vaja iz urejenega razmišljanja pri programiranju. Tale program je tako zoprn, da ga je kar težko v prvem poskusu napisati brez napake.

Do veliko lepše rešitve pridemo z razmislekom. Takole: recimo, da ničti plesalec potrebuje šest korakov, da se vrne na svoje polje. Se pravi, recimo da `do_doma(ples, 0)` (kjer je `ples` pač nek ples, ki ga opazujemo) vrne 6. To pomeni, da bo ta plesalec na svojem začetnem polju po šestih korakih, pa tudi po dvanajstih, osemnajstih, štiriindvajsetih in, vobče, vsakem številu korakov, ki je večkratnik 6.

Poleg tega imamo, recimo, drugega plesalca, ki se vrne na svoje polje po desetih korakih, in zatorej tudi po dvajsetih, tridesetih in vseh ostalih, ki so večkratnik 10.

Kdaj bosta plesalca 0 in 1 naslednjič oba na svojih poljih? Ničti po vseh večkratnikih svojega števila korakov (6) in prvi po vseh večkratnikih svojega števila korakov (10). Če se mi prav zdi, iščemo najmanjši skupni večkratnik 6 in 10, ne?

Pa če imamo še enega plesalca, ki ima do začetnega polja, recimo, 14 korakov? Prav, potem iščemo najmanjši skupni večkratnik 6, 10 in 14; najdemo ga tako, da poiščemo najmanjši skupni večkratnik 6 in 10 (dobimo 30) in potem največji skupni večkratnik 30 in 14, kar bi bilo 210. Če imamo poleg tega še četrtega plesalca ... poiščemo najmanjši skupni večkratnik 210 in njegovega števila korakov.

Za začetek si bomo napisali funkcijo `gcd(a, b)`, ki izračuna največji skupni delitelj `a` in `b` (s tem se sicer ukvarjamo v neki drugi nalogi) in z njeno pomočjo `lcm(a, b)`, ki vrne najmanjši skupni večkratnik. Nadaljujemo pa, kot smo odkrili zgoraj.

```
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

def lcm(a, b):
    return a * b // gcd(a, b)

def vsi_doma(ples):
    korakov = 1
    for e in range(len(ples)):
        korakov = lcm(korakov, do_doma(ples, e))
    return korakov
```

Funkcija bo še malenkost preprostejša, če se zavemo, da seznam ples vedno vsebuje vsa števila od `0` do `len(ples) - 1)`. Drugače ne more biti, saj bi sicer ostalo kakšno polje med plesom prazno, to pa se ne more zgoditi, saj bi prišla na neko drugo polje dva plesalca. Zanko lahko tedaj zapeljemo kar prek `ples` namesto prek `range(len(ples))`. Preiskala bo ista polja, le v drugem vrstnem redu.

```
def vsi_doma(ples):
    korakov = 1
    for e in ples:
        korakov = lcm(korakov, do_doma(ples, e))
    return korakov
```

Ker smo računalnikarji in matematiki bratje, povejmo, kaj o tej nalogi pravijo slednji: pravijo, da permutacije sestavljajo grupo in da smo v prvi različici programa, s simulacijo, množilo permutacijo samo s sabo, dokler nismo dobili identitete.

208. Najbogatejši cvet

Gremo prek vseh vrstic, točneje, njihovih koordinat, y, (`for y in range(len(vrt))`) in znotraj vsake prek vseh elementov, koordinat x (`for x in range(len(vrt[y]))`). Če je element vrt[y][x] večji od elementa na najboljših doslej najdenih koordinatah, `vrt[y][x] > vrt[naj_y][naj_x]`, si zapomnimo te koordinate kot najboljše.

```
def naj_koordinate(vrt):
    naj_x = naj_y = 0
    for y in range(len(vrt)):
        for x in range(len(vrt[y])):
            if vrt[y][x] > vrt[naj_y][naj_x]:
                naj_x, naj_y = x, y
    return naj_x, naj_y
```

Drobnarija, na katere ne smemo pozabiti, je, da na začetku nastavimo `naj_x` in `naj_y` na kako pametno vrednost, recimo kar 0, s čimer za začetek rečemo, da je najboljši zgornji levi cvet.

Paziti moramo še na to, kako delamo s koordinatami: seznam seznamov je obrnjen tako, da imamo *seznam vrstic* (ne *seznama stolpcev*), torej gre prva zanka po y in ne obratno. Tudi do elementa pridemo z `vrt[y][x]` in ne `vrt[x][y]`.

Rešitev je pravilna in kratka. V jezikih, kot sta C in Java, bi reč sprogramirali tako in pustili prevajalniku, naj jo izboljša, če jo zna. V Pythonu bi se za začetek znebili vsaj `range(len(...))`, ob čemer bi program postal tudi nekoliko preglednejši.

Vrstice oštevilčimo z enumerate in tako dobivamo pare indeksov (y) in vrstic. Prav tako oštevilčimo vrstico, da dobivamo pare indeksov (x) in vrednosti.

```
def naj_koordinate(vrt):
    naj_x = naj_y = 0
    for y, vrstica in enumerate(vrt):
        for x, vrednost in enumerate(vrstica):
            if vrednost > vrt[naj_y][naj_x]:
                naj_x, naj_y = x, y
    return naj_x, naj_y
```

Da ne bomo stalno hodili gledat elementa `vrt[naj_y][naj_x]`, si lahko shranimo tudi njegovo vrednost.

```
def naj_koordinate(vrt):
    naj_x = naj_y = 0
    naj_vred = vrt[0][0]
    for y, vrstica in enumerate(vrt):
        for x, vrednost in enumerate(vrstica):
            if vrednost > naj_vred:
                naj_x, naj_y, naj_vred = x, y, vrednost
    return naj_x, naj_y
```

209. Pravilna pot

Tole niti ni prav zares naloga iz seznamov, bolj iz zank.

Pri nalogi Trdnjava smo videli, kako najpreprosteje slediti gibanju po koordinatnem sistemu. Ta naloga je podobna, le z drugačnim ciljem in pogojem: pri trdnjavi smo vrnili `True`, čim se je trdnjava vrnila na začeno polje, tu bomo vrnili `False`, čim bo čebela stopila prek ograje.

```
def pravilna_pot(pot, vrt):
    premiki = {"D": (0, 1), "U": (0, -1), "L": (-1, 0), "R": (1, 0)}
    visina = len(vrt)
    sirina = len(vrt[0])
    x = y = 0
    for korak in pot:
        dx, dy = premiki[korak]
        x += dx
        y += dy
        if not (0 <= x < sirina and 0 <= y < visina):
            return False
    return True
```

210. Dobiček na poti

Seštevanje količine nabranega nektarja je podobno pravilnosti poti, le da zamenjamo pogoj s seštevanjem. Kar poglejte, kako podobni sta si funkciji!

```
def dobicek_na_poti(pot, vrt):
    premiki = {"D": (0, 1), "U": (0, -1), "L": (-1, 0), "R": (1, 0)}
    dobicek = vrt[0][0]
    x = y = 0
    for korak in pot:
        dx, dy = premiki[korak]
        x += dx
        y += dy
        dobicek += vrt[y][x]
    return dobicek
```

Ne smemo pozabiti na prvi (ali na zadnji) cvet: če čebela naredi pet korakov, obere šest cvetov. Da bomo to pravilno upoštevali, smo poskrbeli s tem, da smo na začetku nastavili `dobicek = vrt[0][0]`, nato pa prištevali vrednost vsakega cveta po vsakem čebelinem koraku.

211. Enkratna pot

Spet isto, le da zdaj ne preverjamo pravilnosti in ne seštevamo nektarja, temveč preverjamo, ali je čebela na določenem polju že bila (`if (x, y) in obiskana`). Če je, vrnemo `False`. Če ni, si polje zapomnimo tako, da dodamo terko z njegovimi koordinatami (`(x, y)`) v množico že obiskanih polj.

```
def brez_ponovitve(pot):
    premiki = {"D": (0, 1), "U": (0, -1), "L": (-1, 0), "R": (1, 0)}
    x = y = 0
    obiskana = {(x, y)}
    for korak in pot:
        dx, dy = premiki[korak]
        x += dx
        y += dy
        if (x, y) in obiskana:
            return False
        obiskana.add((x, y))
    return True
```

Funkcije z rešitvami treh nalog – pravilna pot, dobiček na poti in enkratna pot – so praktično enake. Večino funkcije predstavlja koda za sprehajanje po vrtu, vsaka ima le še dve, tri vrstice, ki delajo tisto, kar naj bi funkcija v resnici delala. Ponavljanje večjih kosov kode nikoli ni dobra ideja. Bi lahko tisto, kar se v teh funkcijah ponavlja, spravili v ločeno, skupno pomožno funkcijo?

Napišimo funkcijo `prehodi(pot)`, ki vrne seznam koordinat, prek katerih gre čebela, če leti po podani poti.

```
def prehodi(pot):
    premiki = {"D": (0, 1), "U": (0, -1), "L": (-1, 0), "R": (1, 0)}
    x = y = 0
    koordinate = [(x, y)]
    for korak in pot:
        dx, dy = premiki[korak]
        x += dx
        y += dy
        koordinate.append((x, y))
    return koordinate
```

Zdaj v ostalih treh funkcijah pišemo le še zanke prek teh koordinat.

```
def pravilna_pot(pot, vrt):
    visina = len(vrt)
    sirina = len(vrt[0])
    for x, y in prehodi(pot):
        if not (0 <= x < sirina and 0 <= y < visina):
            return False
    return True

def dobicek_na_poti(pot, vrt):
    dobicek = 0
    for x, y in prehodi(pot):
        dobicek += vrt[y][x]
    return dobicek

def brez_ponovitve(pot):
    obiskana = set()
    for x, y in prehodi(pot):
        if (x, y) in obiskana:
            return False
        obiskana.add((x, y))
    return True
```

Lepota tako napisanih funkcij je v tem, da vsaka res dela samo še tisto, kar je zanjo specifično, zato so preglednejše. Vsa skupna koda, ki se ukvarja samo s spreminjanjem koordinat, je umaknjena v pomožno funkcijo.

Tistim, ki jim je vse, kar vidijo v tej zbirki igrača, pokažimo še rešitev, ki uporablja generatorje. Z njimi se izognemo sestavljanju seznama: zanke v gornjih treh funkcijah gredo prek "seznama", ki se sestavlja kar sproti in nikoli ni nikjer shranjen.

```
def prehodi(pot):
    premiki = {"D": (0, 1), "U": (0, -1), "L": (-1, 0), "R": (1, 0)}
    x = y = 0
    yield x, y
    for korak in pot:
        dx, dy = premiki[korak]
        x += dx
        y += dy
        yield x, y
```

Vse tri funkcije je mogoče napisati tudi v enem zamahu, če znamo.

```
def pravilna_pot(pot, vrt):
    return all(0 <= x < len(vrt[0]) and 0 <= y < len(vrt) for x, y in
prehodi(pot))

def dobicek_na_poti(pot, vrt):
    return sum(vrt[y][x] for x, y in prehodi(pot))

def brez_ponovitve(pot):
    return len(set(prehodi(pot))) == len(pot) + 1
```

Kdor jih ne razume, naj se ne vznemirja. Le zadnja je tako preprosta in temelji na tako splošnem triku, da si ga splača zapomniti. Vsa obiskana polja zložimo v množico (`set(prehodi(pot))`). Če je velikost te množice za 1 večja od števila korakov, smo vsako polje obiskali natančno enkrat.

212. Neobrani cvetovi

Naloga je podobna prejšnjim. Nekoliko je lažja, razmisliti pa je, kako obrniti zanko, da bo čebela obrala tako prvi kot zadnji cvet. V spodnji rešitvi smo to storili tako, da postavimo `vrt[x]` na `0`, preden povečamo `x`, po zanki pa postavimo na `0` še `vrt[x]` za zadnji `x`. Tudi preverjanje, ali je cvet že obran, na ta način lepo sodi v začetek zanke, saj preveri tudi cvet, s katerim čebela začne svojo pot.

```
def ostanki(vrt, pot):
    vrt = vrt[:]
    x = 0
    for korak in pot:
        if vrt[x] == 0:
            break
        vrt[x] = 0
        x += korak
    vrt[x] = 0
    return sum(vrt)
```

Na koncu vrnemo vsoto ostanka.

Ker je naloga izrecno prepovedala spreminjanje podanega seznama, ga, še preden začnemo prčkati po njem, skopiramo z `vrt = vrt[:]`.

Nekateri študenti pri reševanju te naloge naredijo zelo zanimivo napako. Pišejo namreč

```
        if korak > 0:
            x += korak
        else:
            x -= korak
```

Tole čebelo vedno premika v desno, saj takrat, ko je `x` negativen, odšteva ta, negativni `x` – kar spet pomeni prištevanje.

213. Žabji skoki

Funkcij, kot je prva, smo napisali že veliko. Tule jo spet obrnimo nekoliko drugače in čisto tako, za vajo, namesto enega slovarja terk naredimo dva slovarja.

```
def zaba(s):
    x = y = 0
    polja = {(0, 0)}
    dy = {"S": 1, "J": -1, "V": 0, "Z": 0}
    dx = {"S": 0, "J": 0, "V": -1, "Z": 1}
    for smer, koliko in s:
        x += koliko * dx[smer]
        y += koliko * dy[smer]
        polja.add((x, y))
    return polja
```

Za drugo funkcijo le pokličemo prvo in za vsak par (x, y) preverimo ali je slučajno izven predpisanih meja. Če je vrnemo `False`; če ni, ne storimo ničesar in šele na koncu vrnemo `True`.

```
def zaba_znotraj(s, max_x, max_y):
    for x, y in zaba(s):
        if x > max_x or y > max_y or x < 0 or y < 0:
            return False
    return True
```

Za ugotavljanje največje razdalje bomo za vsak par x, y izračunali razdaljo in jo primerjali z največjo doslej.

```
from math import sqrt

def naj_razdalja(s):
    naj = 0
    for x, y in zaba(s):
        d = x ** 2 + y ** 2
        if d > naj:
            naj = d
    return sqrt(naj)
```

Obe funkciji sta igrača za vse, ki poznajo generatorje. Prva zahteva, da preverimo ali velja za vse (`all`), da je `0 <= x <= max_x and 0 <= y <= max_y`, pri čemer za `x` in `y` jemljemo koordinate, na katere je skočila žaba (`for x, y in zaba(s)`).

```
def zaba_znotraj(s, max_x, max_y):
    return all(0 <= x <= max_x and 0 <= y <= max_y for x, y in zaba(s))
```

Drugo funkcijo zanima največja (max) razdalja (`sqrt(x ** 2 + y ** 2)`), pri čemer za koordinati spet jemljemo koordinate polj, na katere skoči žaba (`for x, y in zaba(s)`).

```
def naj_razdalja(s):
    return max(sqrt(x ** 2 + y ** 2) for x, y in zaba(s))
```

Če nočemo računati koren brez potrebe, ga lahko izračunamo le za razdaljo, za katero vemo, da je največja.

```
def naj_razdalja(s):
    return sqrt(max(x ** 2 + y ** 2 for x, y in zaba(s)))
```

214. Lov na muhe

Tole je pa bistveno enostavneje, kot je videti: poznamo polja, na katerih je bila žaba (to dobimo s funkcijo iz prejšnje naloge) in polja, na katerih so (bile, ali pa so še, če imajo srečo) muhe. Zanima nas, na koliko mušjih poljih je bila žaba (ali obratno). To pa ni nič drugega kot presek. Prva funkcija mora vrniti njegovo velikost:

```
def muhe(pozicije, zabja_pot):
    return len(pozicije & zaba(zabja_pot))
```

Druga vrača koordinate pojedenih muh, torej sam presek.

```
def pojedene(pozicije, zabja_pot):
    return pozicije & zaba(zabja_pot)
```

Očitno bi lahko tudi funkcijo muhe napisali tako, da bi klicala funkcijo pojedene,

```
def muhe(pozicije, zabja_pot):
    return len(pojedene(pozicija, zabja_pot)
```

Množico koordinat preživelih muh dobimo tako, da od množice koordinat muh odštejemo množico koordinat, ki jih je obiskala žaba.

```
def prezivele(pozicije, zabja_pot):
    return pozicije - zaba(zabja_pot)
```

215. Žabja restavracija

Tokrat gremo prek vseh vrednosti v slovarju (narocila.values()) in seštevamo, kar najdemo v njih. To je spet dokaj običajna "preštevalna" naloga, le malo drugače je obrnjena.

```
def zberi(narocila):
    from collections import defaultdict
    skupaj = defaultdict(int)
    for jedi in narocila.values():
        for jed in jedi:
            skupaj[jed] += 1
    return skupaj
```

216. Poštar iz Hamiltona

Najprej preverjanje zaporedja. To je nekoliko podobno stari nalogi, v kateri smo preverjali zaporedje domin, le pravilo je drugačno: i-ti element seznama mora imeti vsaj eno skupno števko z elementom i + 1. Kadar iščemo "skupne reči", se splača spomniti na množice. Element, torej številko, spremenimo v niz (to je, niz števk), tega pa v množico (množico števk). To storimo za oba elementa in izračunamo presek teh dveh množic, `set(str(s[i])) & set(str(s[i + 1]))`. Če je prazen, ni skupne števke. Reč preverimo za vse `i` do predzadnjega (ki ga primerjamo z zadnjim).

```
def preveri_zaporedje(s):
    for i in range(len(s) - 1):
        if not set(str(s[i])) & set(str(s[i + 1])):
            return False
    return True
```

Malenkost bolj Pythonovska rešitev uporabi `zip`.

```
def preveri_zaporedje(s):
    for x, y in zip(s, s[1:]):
        if not set(str(x)) & set(str(y)):
            return False
    return True
```

Če poženemo program, ki ga je priporočala pognati naloga (ali pa tudi, če ga ne in malo ugibamo), vidimo, da sestavi vse permutacije, vsa možna zaporedja obiskovanja hiš. Poštar bo lahko opravil svojo nalogo, če je vsaj ena od permutacij pravilna.

```
from itertools import permutations
def postar(naslovi):
    for s in permutations(naslovi):
        if preveri_zaporedje(s):
            return True
    return False
```

Poštar je iz Hamiltona, ker išče nekaj, čemur v teoriji grafov pravijo Hamiltonova pot. Računalniku k temu dodamo, da je problem NP-poln, kar pomeni, da za njegovo reševanje nimamo (bistveno) boljši postopkov kot je ta, ki smo ga ubrali tule.

217. Graf hamiltonskega poštarja

Kadar želimo sestaviti (ali pregledati ali narediti karkoli že s) pari elementov seznama, to storimo z dvojno zanko, v tem primeru

```
for hisa1 in stevilke:
    for hisa2 in stevilke:
```

Za vsak par je potrebno preveriti, ali imata kakšen skupen element in ali gre za dve različni številki. Kako to storimo, vemo iz prejšnje naloge.

```
        if set(str(hisa1)) & set(str(hisa2)) and hisa1 != hisa2:
```

Zdaj pa še to, kar pride okoli: v začetku sestavimo prazno množico, vanjo zlagamo pare in to množico na koncu vrnemo.

```
def pari_his(stevilke):
    pari = set()
    for hisa1 in stevilke:
        for hisa2 in stevilke:
            if set(str(hisa1)) & set(str(hisa2)) and hisa1 != hisa2:
                pari.add((hisa1, hisa2))
    return pari
```

Tule smo množice uporabili dvakrat. Enkrat za rezultat – ker je tako zahtevala naloga. Pare bi lahko zlagali tudi v seznam, če bi naloga hotela tako. Drugič smo množice uporabi interno, znotraj funkcije, za preverjanje skupnih števk. Naloga tega sicer ni zahtevala, bilo pa je praktično.

Da ne bi zanemarjali izpeljanih množic, pokažimo še, kako nalogo rešiti z njimi. Le eno vrstico nam bo vzelo:

```
def pari_his(stevilke):
    return {(hisa1, hisa2) for hisa1 in stevilke for hisa2 in stevilke
            if set(str(hisa1)) & set(str(hisa2)) and hisa1 != hisa2}
```

Zdaj pa iz niza naredimo slovar množic (v katerem matematiki prepoznajo graf).

Tu ni kaj filozofirati, posebej ne tistim, ki so vajeni uporabljati slovarje s privzetimi vrednostmi: če je `graf` slovar, ki ga kani funkcija vrniti, mora za vsak par `(hisa1, hisa2)` dodati v množico `graf[hisa1]` element `hisa2`, torej `graf[hisa1].add(hisa2)`.

```
from collections import defaultdict

def graf_stevilk(pari):
    graf = defaultdict(set)
    for hisa1, hisa2 in pari:
        graf[hisa1].add(hisa2)
        graf[hisa2].add(hisa1)
    return graf
```

Če slovarjev s privzetimi vrednostmi ne poznamo (ali, morda, ne maramo – zakaj že?), se funkcija zaplete kot običajno.

```
def graf_stevilk(pari):
    graf = {}
    for hisa1, hisa2 in pari:
        if hisa1 not in graf:
            graf[hisa1] = {hisa2}
        else:
            graf[hisa1].add(hisa2)
        if hisa2 not in graf:
            graf[hisa2] = {hisa1}
        else:
            graf[hisa2].add(hisa1)
    return graf
```

Slovarji s privzetimi vrednostmi so zakon.

218. Pretakanje

Lenuh bi se lahko naloge lotil tako, da imel spremenljivke `a`, `b` in `p`, ki bi shranjevale količino vode v posameznih piskrih. Grdo bi se zafrknil: to vodi v neskončne špagete `if`-ov, ki preverjajo, kdo pretaka kam, koliko je že tam in koliko bi lahko še šlo tja.

Raje uporabimo seznam s tremi elementi, ki pomenijo količino vode v posameznih posodah, `vsebina = [0, 0, 500]`. Da bomo vedeli, kje je kateri, si pripravimo slovar `posode = {"A": 0, "B": 1, "P": 2}`. Poleg tega potrebujemo še terko (lahko bi bil tudi seznam, a jasno povejmo, da je reč fiksna, nespremenljiva) z velikostmi posod: `velikost = (5, 8, 1000)`.

Zdaj v vsaki potezi (`for poteza in s`) najprej ugotovimo indeks posode, iz katere in v katero točimo, `iz = posode[poteza[0]]` in `v = posode[poteza[-1]]`. Če, recimo, točimo iz A v P, bo `iz` enak `0` in `v` bo enak 2.

Z naslednjim `if`-om razlikujemo med tem, ali bomo točili, dokler ne bo ciljna posoda polna ali dokler ne bo začetna prazna. V prvem primeru zmanjšamo količino mleka v začetni za toliko, kolikor je prostora v ciljni, za ciljno pa zabeležimo, da je polna. V drugem primeru pa k ciljni posodi prištejemo količino mleka v začetni, za začetno pa zabeležimo, da je prazna.

Funkcija vrne prva dva elementa seznama `vsebina`, to je, količina mleka v prvih dveh posodah, ki ju, kot se za rezultate funkcij spodobi, spremenimo v terko.

```
def pretakanje(s):
    posode = {"A": 0, "B": 1, "P": 2}
    velikost = (5, 8, 1000)
    vsebina = [0, 0, 500]
    for poteza in s:
        iz = posode[poteza[0]]
        v = posode[poteza[-1]]
        if vsebina[v] + vsebina[iz] > velikost[v]:
            vsebina[iz] -= velikost[v] - vsebina[v]
            vsebina[v] = velikost[v]
        else:
            vsebina[v] += vsebina[iz]
            vsebina[iz] = 0
    return tuple(vsebina[:2])
```

219. Slovar anagramov

Kot običajno je tudi pri tej nalogi koristno poznati slovarje s privzetimi vrednostmi; skrajšalo jo je za tri vrstice flanca. Vse, kar nam je storiti, je iti prek vseh besed in v množico, ki pripada ustreznemu ključu (`"".join(sorted(beseda))`) dodati besedo.

Druga funkcija le vrača množico, ki pripada ključu, ki ga dobimo iz te besede.

```
def slovar_anagramov(besede):
    from collections import defaultdict
    slovar = defaultdict(set)
    for beseda in besede:
        slovar["".join(sorted(beseda))].add(beseda)
    return slovar

def poisci_anagrame(beseda, slovar):
    return slovar["".join(sorted(beseda))]
```

220. Človek ne jezi se

Ta naloga je predvsem vaja iz programerske spretnosti. Rešitev je lahko preprosta, ali, če smo nerodni, strašno zapletena.

```
def clovek_ne_jezi_se(igralcev, meti):
    pozicije = [0] * igralcev
    for poteza, met in enumerate(meti):
        igralec = poteza % igralcev
        nova = pozicije[igralec] + met
        pozicije = [0 if x == nova else x for x in pozicije]
        pozicije[igralec] = nova
    return pozicije
```

V prvi vrstici me zanima elegantno pripravimo seznam iz ničel: `[0] * igralcev`. Nato uporabimo `enumerate`, da oštevilčimo poteze; igralec, ki je na potezi, je `poteza % igralcev`. Če se tega trika ne domislimo, bomo uporabljali vzorec

```
igralec = 0
for met in meti:
    ...
    igralec += 1
    if igralec == igralcev:
        igralec = 0
```

ali

```
igralec = 0
for met in meti:
    ...
    igralec = (igralec + 1) % igralcev
```

Tudi s tem ni nič narobe.

Nato izračunamo novo pozicijo tega igralca. Trenutne `pozicije` zamenjamo z novo tabelo, v katero vse tiste elemente, ki so enaki `nova` zamenjamo z `0` – to so tisti, ki morajo na začetek. Seveda se da to narediti tudi na bolj zapleten način.

Končno zares prestavimo še tega igralca, `pozicije[igralec] = nova`. Če bi ga prestavili že prej, bi ga vrstico višje vrnili na začetek, kot da je "požrl" samega sebe.

Kot vidimo, ne gre za preveč zapleteno reč, pač pa je lahko zelo zoprna in dolga, če si nekoliko neroden.

221. Največ dvakrat

Ker moramo spreminjati seznam, je potrebno uporabljati `del` ali `pop` – ne moremo kar tako sestaviti novega seznama. Pri tem pa imamo težavo: če gremo z zanko `for` čez seznam, je brisanje elementov znotraj zanke zelo slaba ideja. Ne deluje. Primere smo videli.

Drugo, kar zahtevala naloga, je, da vodimo evidenco o tem, kolikokrat se je določen element že pojavil. Tu pomaga slovar, ali, še boljše, `defaultdict`.

Rešitev je lahko, recimo ta

```
from collections import defaultdict

def najvec_dve(s):
    kolikokrat = defaultdict(int)
    i = 0
    while i < len(s):
        e = s[i]
        kolikokrat[e] += 1
        if kolikokrat[e] > 2:
            del s[i]
        else:
            i += 1
```

Namesto `for` uporabimo `while`. Če element pobrišemo, ne povečamo števca, saj se bo naslednji element premaknil na mesto trenutnega.

Preprosteje je sestaviti nov seznam, katerega elemente na koncu prepišemo v `s`.

```
def najvec_dve(s):
    t = []
    for e in s:
        if t.count(e) < 2:
            t.append(e)
    s[:] = t
```

Ta rešitev sicer ni najhitrejša, ker uporablja `count`. Lepše bi bilo spet uporabiti `defaultdict`, vendar ... takole je pa krajše.

Ena od študentk se je spomnila domiselne rešitve. Preštela je viške, obrnila seznam, odstranila viške in ga spet obrnila. K njeni rešitvi lahko dodamo še `Counter` in tako dobimo:

```
from collections import Counter
def najvec_dve(s):
    s.reverse()
    for n, r in Counter(s).items():
        for i in range(r - 2):
            s.remove(n)
    s.reverse()
```

222. Seštej zaporedne

Tale naloga je nekoliko sitna, saj nima namreč kakšne očitne, lepe rešitve, pa še na par zoprnih detajlov je potrebno paziti.

Tule je ena od možnih rešitev.

```
def sestej_zaporedne(s):
    vsote = []
    trenutna = None
    prejsnji = None
    for e in s:
        if e == prejsnji:
            trenutna += e
        else:
            if trenutna is not None:
                vsote.append(trenutna)
            trenutna = e
        prejsnji = e
    if trenutna is not None:
        vsote.append(trenutna)
    return vsote
```

V `vsote` zlagamo, kar bomo vrnili. `trenutna` pove vsoto trenutnega zaporedja enakih elementov, `prejsnji` pa je prejsnji element.

Zapeljemo se čez seznam. Če je trenutni element (`e`) enak prejšnjemu, le povečamo trenutno vsoto. Če ni in če trenutna vsota ni `None` (to je, če nismo na začetku seznama), v `vsote` damo trenutno vsoto. V vsakem primeru pa začenjamo šteti od začetka, torej je trenutna vsota enaka `e`.

Ko je vse končano, v vsote dodamo še trenutno vsoto, torej tisto, ki je ni "prekinil" noben element več. Vendar le, če le-ta obstaja; trenutne vsote ni, kadar je seznam, ki smo ga dobili kot argument, prazen.

Precej elegantnejša rešitev opazuje pare zaporednih elementov, za kar uporabimo znano frazo `zip(s, s[1:])`. Če je trenutni element enak prejšnjemu, ga prištejemo k zadnji vsoti, sicer v seznam vsot dodamo trenutni element.

```
def sestej_zaporedne(s):
    vsote = s[:1]
    for prej, zdaj in zip(s, s[1:]):
        if zdaj == prej:
            vsote[-1] += zdaj
        else:
            vsote.append(zdaj)
    return vsote
```

Seznam vsot, ki ga bomo vrnili, zdaj v začetku ni prazen, temveč že vsebuje prvi element – v izogib sitnostim z zadnjim. To bi sicer lahko dosegli `vsote = [s[0]]`, vendar to ne bo delovalo, če je seznam `s` morda prazen. Zato pišemo `vsote = s[:1]`; to deluje tudi za prazne sezname – če je `s` prazen seznam, bo prazen tudi seznam `vsote`.

223. Intervali

Naloga je zoprna, ker Python ne ponuja ničesar, kar bi jo olajšalo – množice, rezine, slovarji, nič od tega nas ne reši. Začnimo torej z mukotrpnim pregledovanjem, kakršnega bi bili (stalno, veliko pogosteje kot v Pythonu) deležni v bolj zateženih jezikih.

```
def intervali(s):
    res = []
    zac = 0
    while zac < len(s):
        kon = zac + 1
        while kon < len(s) and s[kon] == s[kon - 1] + 1:
            kon += 1
        res.append((s[zac], s[kon - 1]))
        zac = kon
        return res
```

Gremo po seznamu; `zac` bo začetek zaporedja elementov. Pri vsakem začetku poiščemo, z notranjo zanko, ustrezan konec; `kon` bo indeks elementa za zadnjim elementom zaporedja. Ko poznamo začetek in konec, dodamo v seznam, ki ga bomo vrnili, par `(s[zac], s[kon - 1])`. Novi začetek bo, kjer je konec tega.

Notranja zanka teče, dokler še nismo na koncu seznama in je naslednji element za 1 večji od prejšnjega. Zunanja teče do takrat, ko bi bil začetek izven seznama.

V takšnih primerih nam pogosto pomaga `zip`. Tule pa pravzaprav ne, gornja rešitev je lepša od rešitve z `zip`om:

```
def intervali(s):
    if not s:
        return []
    ints = []
    f = s[0]
    for e1, e2 in zip(s, s[1:]):
        if e2 != e1 + 1:
            ints.append((f, e1))
            f = e2
    ints.append((f, e2))
return ints
```

`f` je prva vrednost zaporedja, `e1` zadnja in `e2` prva vrednost naslednjega zaporedja. Spremenljivki `e1`, `e2` gresta prek zaporednih parov, dokler ne pridemo do mesta, ko se ne razlikujeta za 1. Takrat shranimo meje trenutnega zaporedja, `(f, e1)`, in začnemo novo, `f = e2`. Kar zoprna reč.

Drugi del naloge, `razpisi` je preprost.

```
def razpisi(intervali):
    res = []
    for a, b in intervali:
        res += range(a, b + 1)
    return res
```

224. Dolžine ladij

Funkcija ne ve, ali je ladja navpična ali vodoravna. Vendar naj nas to ne ustavi: izmerimo ladjino širino in višino, potem pa vrnemo, kar od tega je večje.

```
def dolzina(plosca, x, y):
    v = 1
    while x + v < len(plosca[y]) and plosca[y][x + v] == "X":
        v += 1
    s = 1
    while y + s < len(plosca) and plosca[y + s][x] == "X":
        s += 1
    return max(v, s)
```

Da izvemo velikost največje ladje, izmerimo vse ladje na vseh poljih in vrnemo največje število. To lahko naredimo počasi (kar je tako trivialno, ad bomo kar preskočili) ali z generatorjem (kar je zanimivo, zato pokažimo).

```
def najdaljsa_ladja(plosca):
    return max(dolzina(plosca, x, y)
               for x in range(len(plosca[0])) for y in range(len(plosca))
               if plosca[y][x] == "X")
```

Premetavanje nizov in besedil

225. Sekunde

Prva naloga je najpreprostejša, le `split` je potrebno pravilno poklicati, pa nize moramo znati pretvoriti v številke in jih prav namnožiti v sekunde.

```
def cas_v_sekunde(s):
    h, m, s = s.split(":")
    return 3600 * int(h) + 60 * int(m) + int(s)
```

Pravzaprav je druga še preprostejša. Težavo predstavlja le tem, ki ne znajo oblikovati izpisa. Poleg tega je potrebno le še premisliti, kako pravilno deliti in računati ostanke. (Kdor je vajen starega Pythona, ga bo treba spomniti, da je potrebno za celoštevilsko deljenje uporabiti `//`.)

```
def sekunde_v_cas(s):
    return "%02i:%02i:%02i" % (s // 3600, s % 3600 // 60, s % 60)
```

Če malo pomislimo, pa je najpreprostejša tretja. No, tretja naloga je nespodobno težka za vse, ki jim ne pride na misel, da bi uporabili gornji funkciji. Izračunati, koliko je od enajst minut čez deset do pet minut čez dvanajst je res zoprno, če ločeno odštevamo sekunde, minute in ure. Veliko lažje je pretvoriti časa v sekunde, ju odšteti in potem pretvoriti nazaj v človeški zapis.

```
def razlika_casov(s1, s2):
    return sekunde_v_cas(cas_v_sekunde(s2) - cas_v_sekunde(s1))
```

226. Naslednji avtobus

Naloga je klasičen "poišči najmanjši element" z dodatnim trikom: kako primerjati številke avtobusov? Vsekakor jih bomo morali iz nizov pretvoriti v števila, da bo 2 manj kot 11. Po drugi strani pa se je potrebno znebiti odvečne črke na koncu. Takole naredimo: uporabili bomo celo ime, če je zadnji znak imena avtobusa števka (to nam pove metoda `isdigit`), sicer pa vse do zadnjega znaka. Ali, z dobesednim prevodom v Python, hočemo

```
avtobus if avtobus[-1].isdigit() else avtobus[:-1]
```

Zdaj predpostavimo, da je do naslednjega avtobusa še en dan (1440 minut). Gremo prek vseh avtobusov in njihovih prihodov. Za vsak avtobus z gornjim izrazom dobimo njegovo številko in jo z `int` pretvorimo iz niza v število. Poleg tega s funkcijo min poiščemo prvi prihod avtobusa s to številko. Če bo ta avtobus prišel pred prvim, ki smo ga našli doslej, ali pa pride istočasno, vendar ima nižjo številko, si to zapomnimo. Točneje, zapomniti si moramo čas prihoda, številko (kot število, zato da bomo lažje primerjali) in pravo oznako proge (npr. 6b).

```
def naslednji_avtobus(prihodi):
    kdaj = 1440
    for avtobus, minute in prihodi.items():
        stev = int(avtobus if avtobus[-1].isdigit() else avtobus[:-1])
        minute = min(minute)
        if minute < kdaj or minute == kdaj and stev < najstev:
            kdaj, najstev, najavtobus = minute, stev, avtobus
    return najavtobus
```

227. Eboran

Niz razbijemo na besede (`split`), vsako besedo posebej obrnemo in to zlagamo v nov seznam, ki ga spet združimo (`join`).

```
def eboran(stavek):
    s = []
    for bes in stavek.split():
        s.append(bes[::-1])
    return " ".join(s)
```

Nekoliko krajše je, če besede kar sproti zlagamo v nov niz.

```
def eboran(stavek):
    s = ""
    for b in stavek.split():
        s += b[::-1] + " "
    return s[:-1]
```

Pri vračanju rezultata odbijemo zadnjo črko, da se znebimo odvečnega presledka na koncu. ("Beh, ceneno," pravijo vsi Cjevski programerji, ki se ubijajo z zastavicami za izpisovanje seznamov, ločenih z vejicami.)

Tudi rešitev z izpeljevanjem seznamov ni težka, čeprav je kratka.

```
def eboran(stavek):
    return " ".join(x[::-1] for x in stavek.split())
```

Dodatna naloga je zanimiva zato, ker jo je težko rešiti drugače kot z enim zamahom. Potrebno je poznavanje regularnih izrazov; brez njih je zoprno (poskusite, spominja na Najdaljše nepadajoče zaporedje!), z njimi pa trivialno.

```
import re
def eboran2(stavek):
    return re.sub("\w*", lambda x:x.group()[::-1], stavek)
```

228. Cenzura

Najprej očitna rešitev. Pripravimo prazen niz za novo besedilo. Podano besedilo razbijemo na besede (`split`) in gremo prek njih. Če je beseda prepovedana, v novo besedilo dodamo "besedo" s toliko X-i, kot je treba, sicer pa prepišemo originalno besedo. Pri preverjanju ne

smemo napisati `if beseda in prepovedana`, saj naloga zahteva, naj bo beseda prepovedana tudi, če je napisana z velikimi črkami. Ker vemo, da so vse prepovedane besede podane z malimi črkami, bomo v pogoju namesto `beseda` pisali `beseda.lower()`, pa bo.

V vsakem primeru dodamo na konec še presledek, da se nam besede ne zlepijo skupaj. Ker bo imel niz tako na koncu odvečen presledek, vrnemo le niz do predzadnjega znaka.

```
def cenzura(besedilo, prepovedane):
    novo = ""
    for bes in besedilo.split():
        if bes.lower() in prepovedane:
            novo += "X" * len(bes) + " "
        else:
            novo += bes + " "
    return novo[:-1]
```

Celotno funkcijo lahko precej skrajšamo, če zamenjamo pogojni stavek z izrazom, ki vrne X-e, če je beseda prepovedana, sicer pa besedo, ali, v Pythonu, `"X" * len(bes) if bes.lower() in prepovedane else bes`.

```
def cenzura(besedilo, prepovedane):
    novo = ""
    for bes in besedilo.split():
        novo += ("X" * len(bes) if bes.lower() in prepovedane else bes) + " "
    return novo[:-1]
```

Zdaj pa se vendarle spomnimo na `join`, ki nam poenostavi delo – sploh, če poznamo izpeljane sezname.

```
def cenzura(besedilo, prepovedane):
    return " ".join(
        ("X"*len(bes) if bes.lower() in prepovedane else bes)
         for bes in besedilo.split())
```

Če hočemo sestaviti še funkcijo, ki upošteva tudi ločila, bomo težko preživeli brez regularnih izrazov. Z njimi pa nam ne bo hudega.

```
import re

def cenzura(besedilo, prepovedane):
    return re.sub("\w+", lambda mo: "X" * len(mo.group()) if
        mo.group().lower() in prepovedane else mo.group(), besedilo)
```

229. Črkovalnik

V funkciji `crka` se moremo spotakniti kvečjemu ob drugi pogoj in to, kar mu sledi. Ne smemo pozabiti, da je `a` niz, zato ga moramo primerjati z nizi (`"0" <= a <= "9"` in ne `0 <= a <= 9`), ko množimo z njim, pa ga moramo pretvoriti v število (`c * int(a)` in ne `c * a`).

```
def crka(c, a):
    if a == "U":
        return c.upper()
    if "0" <= a <= "9":
        return c * int(a)
    if a == "x":
        return ""
    if a == ".":
        return c
```

V drugi funkciji moramo istočasno prek dveh nizov. To smo vzeli že pri drugih nalogah: ne bomo pisali

```
for c in beseda:
    for a in koda:
```

temveč spretno uporabimo zip:

```
def kodiraj(beseda, koda):
    r = ""
    for c, a in zip(beseda, koda):
        r += crka(c, a)
    return r
```

Če še vedno ne vemo za zip, potrebujemo indekse, recimo

```
def kodiraj(beseda, koda):
    r = ""
    for i in range(len(beseda)):
        r += crka(beseda[i], koda[i])
    return r
```

Resnejšega programerja v Pythonu zmoti prištevanje v `r`; zamenjal ga bo z `join`om.

```
def kodiraj(beseda, koda):
    return "".join(crka(c, a) for c, a in zip(beseda, koda))
```

230. Hosts

Program ni vreden komentarja, le navodilom iz naloge sledi.

```
def beri_hosts():
    d = {}
    for line in open("hosts"):
        if "#" in line:
            line = line[:line.find("#")]
        line = line.strip()
        if line:
            ip, name = line.split()
            d[name] = ip
    return d
```

231. Uporabniške skupine

Gre predvsem za vajo v uporabi funkcije `split` in malo prekladanja slovarjev.

```
import collections
def beri_skupine(ime_dat):
    users = collections.defaultdict(list)
    for line in open(ime_dat):
        group_name, password, group_id, group_list = line.strip().split(":")
        for user in group_list.split(","):
            users[user].append(group_name)
    return dict(users)
```

Nekaj rokohitrstva smo si privoščili pri uporabi metode `split`: ker vemo, da bomo dobili seznam s štirimi elementi, smo ga kar takoj odpakirali v štiri spremenljivke.

232. Skrito sporočilo

Nalogo bi lahko reševali tako, da bi iskali zaporedja `". "` z metodo `find`, ki ji lahko podamo tudi, od katerega znaka naprej naj išče. Z malo elegance to storimo takole

```
def razkrij(skrito):
    razkrito = skrito[0]
    z = 0
    while True:
        z = skrito.find(". ", z+1)
        if z == -1:
            return razkrito
        else:
            razkrito += skrito[z+2]
```

Najprej poberemo prvo črko skritega sporočila. Spremenljivka `z` bo predstavljala mesto zadnje pike; v začetku jo postavimo kar na `0`. Nato v nedogled ponavljamo tole: poiščemo naslednjo piko (iščemo od znaka `z+1` naprej) in rezultat, mesto nove pike, shranimo v `z`. Če nove pike ni (`z==-1`), vrnemo razkrito sporočilo. Sicer pa v razkrito sporočilo prepišemo drugi znak za piko (na mestu `z` je pika in na mestu `z+1` presledek) in nadaljujemo.

Še elegantnejša in bolj Pythonovska rešitev (možna bi bila tudi v drugih skriptnih jezikih, recimo phpju) je razdeliti niz glede na podniz `". "`. Razkrito sporočilo so prvi znaki dobljenih koščkov.

```
def razkrij(skrito):
    razkrito = ''
    for c in skrito.split('. '):
        razkrito += c[0]
    return razkrito
```

Odtod pa ni daleč do

```
def razkrij(skrito):
    return "".join(c[0] for c in skrito.split('. '))
```

233. Iskanje URLjev

Za takšne reči navadno uporabljamo nekaj, čemur se učeno pravi regularni izrazi. Tule se bomo delali, da jih ne znamo (najbrž jih tudi res ne). Nalogo lahko kar preprosto rešimo tako, da uporabimo funkcijo `split`, da dobimo vse besede. Vrnemo tiste, ki se začnejo s http:// ali https://.

```
def najdi_URLje(s):
    return [m for m in s.split() if m.startswith(("http://", "https://"))]
```

Če se izognemo tudi temu, pa imamo priložnost za prijetno programersko vajo. Poglejmo.

```
def najdi_URLje(s):
    URLji = []
    for tip in ["http://", "https://"]:
        zac = 0
        while True:
            zac = s.find(tip, zac)
            if zac == -1:
                break
            else:
                kon = len(s)
                for znak in " \t\n\r":
                    kon2 = s.find(znak, zac)
                    if kon2 != -1 and kon2 < kon:
                        kon = kon2
                URLji.append(s[zac:kon])
                zac = kon
    return URLji
```

Naslove oblike http:// in https:// bomo iskali ločeno, najprej ene, potem druge. Temu je namenjena zunanja zanka. Z njo za vsak tip ponovimo naslednje:

Kar sledi, je podobno temu, kar smo s `find` počeli v nalogi Skrito sporočilo. Spremenljivka `zac` vsebuje mesto v nizu, od katerega naprej iščemo. Z `s.find(tip, zac)` bomo iskali prvi `http://` ali `https://` za mestom `zac`. V `zac` bomo v začetku vpisali `0`, da se iskanje začne na začetku; `find` bo vanj prepisal mesto, kjer se začne naslednji URL, na koncu zanke pa ga z `zac = kon` postavimo za zadnji najdeni URL.

Zanko `while` ponavljamo v neskončnost, a jo prekinemo, ko ne najdemo nobenega URLja več (`zac == -1`). Ker nismo več popolni začetniki, vemo, da bo ta `break` prekinil le najbolj notranjo zanko, v kateri se nahaja, torej zanko `while`, ne pa zanke `for`. Če najdemo URL, pa je treba poiskati njegov konec. Konec bomo zapisali v `kon`; v začetku predpostavimo, da je URLja konec kar, ko je konec niza, `kon = len(s)`. Nato pa poskušamo najti katerega od znakov, s katerimi bi se lahko končal prej – presledek, tabulator, konec vrstice... Tu bi lahko dodali še kaj, česar URL ne more vsebovati. Za vsakega od teh poiščemo njegovo prvo pojavitev (za mestom `zac`); če se znak pojavi in če se pojavi pred trenutno predvidenim koncem (`if kon2 != -1 and kon2 < kon`), bo to novi predvideni konec (`kon = kon2`). Ko je to za nami, dodamo URL v seznam URLjev (`s[zac:kon]`) in postavimo začetek naslednjega iskanja za konec trenutnega URLja, (`zac = kon`).

234. Deli URLja

Začetnik bi storil takole. Najprej bi poiskal podniz ://. Kar je pred njim, je protokol. Kar je za njima bo mrcvaril naprej: poiskal bo prvo poševnico. Kar je pred njo, je strežnik, kar za njo (morda tudi nič) je pot.

```
def razbijURL(s):
    pp = s.find("://")
    protokol, ostalo = s[:pp], s[pp + 3:]
    if "/" in ostalo:
        pp = ostalo.find("/")
        streznik, pot = ostalo[:pp], ostalo[pp + 1:]
    else:
        streznik, pot = ostalo, ""
    return protokol, streznik, pot
```

Resen programer brez pomišljanja uporabi regularne izraze. Samo tokrat poglejmo, kako bi bilo to videti:

```
import re
re_URL = re.compile("(\w+)://([^/]+)/?(.*)")
def razbijURL(url):
    return re_URL.match(url).groups()
```

235. Trgovinski računi

Rešitev je preprostejša, kot se zdi. Beremo vrstice datoteke in jih `split`amo s tabulatorjem. Ker naloga obljublja, da se bo tabulator pojavil le v vrsticah s številkami, bo imel dobljeni seznam dva elementa natanko tedaj, ko vrstica vsebuje izdelek in ceno.

Če je "izdelek" v resnici "skupaj", preverimo, ali je vsota izdelkov enaka številu, ki je v drugem elementu (razbite) vrstice. Primerjanje bo vrnilo `True` ali `False`. Sicer pa k vsoti prištejemo ceno trenutnega izdelka.

Če se zanka izteče, to pomeni, da nismo naleteli na vrstico "skupaj" in vrniti je potrebno `False`.

```
def preveri_racun(ime_dat):
    vsota = 0
    for l in open(ime_dat):
        s = l.split("\t")
        if len(s) == 2:
            if s[0] == "skupaj":
                return vsota == float(s[1])
            else:
                vsota += float(s[1])
    return False
```

236. Izračun računa

Najprej bomo prebrali cenik v slovar, katerega ključi bodo kode, vrednosti pa cene. Vsako vrstico datoteke razbijemo s `split`, prvi element, kodo, pustimo kot niz, drugega pretvorimo v število (`float`) in zapišemo v slovar.

Nato beremo drugo datoteko, seznam nakupljenih izdelkov. Vsako vrstico spet razbijemo. Kodo uporabimo zato, da iz slovarja dobimo ceno, pomnožimo pa jo z drugo številko iz vrstice, ki jo moramo spet pretvoriti v število.

```
def racunaj(cene, nakupi):
    cenik = {}
    for izdelek in open(cene):
        koda, cena = izdelek.split()
        cenik[koda] = float(cena)
    vsota = 0
    for izdelek in open(nakupi):
        koda, kolicina = izdelek.split()
        vsota += cenik[koda] * float(kolicina)
    return vsota
```

Bolj izkušen programer bi cenik prebral v enem zamahu, z

```
cenik = {koda: float(cena) for koda, cena in (s.split() for s in open(cene))}
```

Celotno funkcijo bi prepisal – kar pa ni nujno najlepše – v

```
def racunaj(cene, nakupi):
    cenik = {koda: float(cena) for koda, cena in (s.split() for s in open(cene))}
    return sum(cenik[koda] * float(kolicina)
               for koda, kolicina in (s.split() for s in open(nakupi)))
```

237. Numerologija

Vrednost črke `c` dobimo tako, da od `ord(c)` odštejemo 64. Vrednosti seštejemo, nato pa ponavljamo tole: gremo preko števk vsote, jih seštevamo... in to počnemo, dokler vsota ni manjša od 9.

```
def numerologija(ime):
    vs = 0
    for c in ime.upper():
        vs += ord(c) - 64
    while vs > 9:
        nova = 0
        for c in str(vs):
            nova += int(c)
        vs = nova
    return vs
```

Kot druge tudi ta naloga postane smešno preprosta z malo funkcijskega programiranja.

```
def numerologija(ime):
    s = sum(map(ord, ime.upper())) - 64*len(ime)
    while s > 9:
        s = sum(map(int, str(s)))
    return s
```

Mimogrede smo se domislili še, da nam 64 ni potrebno odšteti vsakič posebej, temveč ga lahko odštejemo na koncu za vse črke skupaj.

238. Grep

Gre za tipičen primer naloge, ki jo je preprosteje reševati v skriptnih jezikih, ki so navadno boljše opremljeni za premetavanje besedil kot prevajani jeziki.

```
import glob
def grep(mask, substr):
    res = []
    for f in glob.glob(mask):
        if substr in open(f).read():
            res.append(f)
    return res
```

Tole spet kar kliče po krajši rešitvi.

```
def grep(mask, substr):
    return [f for f in glob.glob(mask) if substr in open(f).read()]
```

239. Preimenovanje datotek

Opis naloge je precej daljši od primerno spretno sestavljene rešitve. (To je sicer odvisno od spretnosti reševalca; nekateri študenti so napisali rešitev, ki je imela več kot sto vrstic.)

```
import os

extensions = [".avi", ".mpg", ".mkv", ".rm", ".mp4"]
notCapital = ["a", "an", "the", "above", "against", "along", "alongside", "amid",
"amidst", "around", "as", "aside", "astride", "at", "athwart", "atop", "before",
"behind", "below", "beneath", "beside", "besides", "between", "beyond", "but",
"by", "down", "during", "for", "from", "in", "inside", "into", "of", "on", "onto",
"out", "outside", "over", "per", "plus", "than", "through", "throughout", "till",
"to", "toward", "towards", "under", "underneath", "until", "upon", "versus",
"via", "with", "within", "without"]
```

```
for fname in os.listdir("."):
    fname = fname.lower()
    base, ext = os.path.splitext(fname)
    if ext not in extensions:
        continue
    base = base.replace(".", " ")
    words = base.split()
    for i in range(len(words)):
        if i==0 or words[i-1]=="-" or words[i] not in notCapital:
            words[i] = words[i].capitalize()
    base = " ".join(words)
    os.rename(fname, base+ext)
```

Najprej spremenimo vse črke v male črke in razbijemo ime datoteke na osnovo in končnico. Če datoteka nima ustrezne končnice, nadaljujemo (`continue`) z naslednjo datoteko.

Če je končnica prava, pa zamenjamo vse pike s presledki in nato razbijemo ime na besede. Nato se zapeljemo čez vse besede in damo veliko začetnico tistim, ki jim je treba dati veliko začetnico. In katere so to? To so prva beseda (`i==0`), beseda, pred katero je bil pomišljaj (`words[i-1]=="-"`) in tudi vse druge besede, razen tistih s seznama (`words[i] not in notCapital`).

Ko je to postorjeno, nam preostane le še, da združimo besede nazaj v ime (`join`) in preimenujemo datoteko.

240. Vse datoteke s končnico .py

Funkcija ne bo prejela le enega argumenta, temveč dva. Prvi bo predstavljal začetni direktorij, drugi pa poddirektorij znotraj njega. Funkcija bo preiskala ta poddirektorij; vse datoteke bo dodala v seznam, ko bo naletela na poddirektorij, pa se bo rekurzivno poklicala, pri čemer bo drugi argument dopolnila z imenom poddirektorija.

```
import os
def koncnica_py(dir, subdir=""):
    sez = []
    for ime in os.listdir(os.path.join(dir, subdir)):
        if os.path.isdir(os.path.join(dir, subdir, ime)):
            sez += koncnica_py(dir, os.path.join(subdir, ime))
        elif ime.endswith(".py"):
            sez.append(os.path.join(subdir, ime))
    return sez
```

Kot vse rekurzivne funkcije je tudi ta dokaj preprosta, vendar jo je težko opisati; lažje jo boste razumeli, če boste razmišljali ob pogledu na kodo.

Nadležna reč te funkcije so stalni `os.path.join`i. V dobršni meri se jim izognemo, če namesto argumentov, ki povedo direktorij in poddirektorij (ki ju je potem potrebno stalno združevati), povemo direktorij in število znakov poddirektorija, ki pripadajo začetnemu direktoriju in jih zato ni potrebno vključiti v ime. Da bo funkcija takšna, kot jo zahteva naloga, bo imel ta

argument privzeto vrednost `None`; če ne bo podan, kar bom vedeli po tem, da je njegova dolžina 0 (in to se bo zgodilo le, ko bo funkcija klicana od zunaj), ga bomo zamenjali z dolžino začetnega direktorija in še ena zraven, da odbijemo še prvo ločilo (`/` oz `\`, odvisno od tega, ali smo na Linuxu ali Windowsih).

```
import os
def koncnica_py(dir, zacd=0):
    sez = []
    for ime in os.listdir(dir):
        celoime = os.path.join(dir, ime)
        if os.path.isdir(celoime):
            sez += (koncnica_py(celoime, zacd or len(dir) + 1))
        elif celoime.endswith(".py"):
            sez.append(celoime[zacd:])
    return sez
```

Pythonov modul `os` ima funkcijo `walk`. Kaj počne, si oglejte v dokumentaciji. Z njo si lahko precej poenostavimo rešitev naloge, saj funkcija sama poskrbi za rekurzivni del.

```
def koncnica_py(dir):
    sez = []
    for subdir, dirs, files in os.walk(dir):
        for ime in files:
            if ime.endswith(".py"):
                sez.append(os.path.join(subdir[len(dir):], ime))
    return sez
```

241. Uredi CSV

Najprej rešimo nalogo z začetniško potrpežljivostjo.

```
import os
def uredi_csv(ime):
    novo_ime = "%s_sorted%s" % os.path.splitext(ime)
    urejena = open(novo_ime, "wt")
    for vrstica in open(ime):
        besede = vrstica.strip().split(",")
        besede.sort()
        urejena.write(", ".join(besede) + "\n")
```

Gornje je mogoče stisniti v eno vrstico ... ki ni podobna ničemur. Pokažimo, da se da, razlagali in promovirali pa takšnega programiranja ne bomo.

```
open("%s_sorted%s" % os.path.splitext(ime), "wt").write(
    "".join(", ".join(sorted(v.strip().split(","))) + "\n"
    for v in open(ime)))
```

242. Zaporedni samoglasniki in soglasniki

Predpostavimo, da so besede ločene s presledki, drugih ločil ni. Niz ločimo na besede in za vsako posebej pogledamo, ali ne vsebuje dveh zaporednih samoglasnikov ali štirih zaporednih soglasnikov. Z `break` v notranji zanki poskrbimo, da vsako besedo dodamo le enkrat.

```
def zaporedni(s):
    samoglasniki = set("aeiouAEIOU")
    besede = []
    for beseda in s.split():
        for i in range(len(beseda)):
            if i-ti in i+1-vi znak sta samoglasnika, ali pa so znaki od i-tega do i+4-ga soglasniki :
                besede.append(beseda)
                break
    return besede
```

Manjka še pogoj. Duhamorna različica je očitna:

```
            if       i < len(beseda)-1
                  and beseda[i] in samoglasniki \
                  and beseda[i + 1] in samoglasniki \
               or     i < len(beseda)-3
                  and beseda[i] not in samoglasniki
                  and beseda[i + 1] not in samoglasniki \
                  and beseda[i + 2] not in samoglasniki \
                  and beseda[i + 3] not in samoglasniki:
```

Za bruhat. Čisto prvega pogoja se znebimo, če spustimo `i` samo do predzadnjega znaka besede, a še vedno nam razvlečeni pogoj ni všeč. Priznati je sicer treba, da je takšna rešitev preprosta in hitrejša od naslednje, vendar je neprivlačna.

Lepšo rešitev spet nudijo regularni izrazi.

```
import re
def zaporedni(s):
    return [mo.group() for mo in
            re.finditer(r"\w*([aeiou]{2}|[bcdfghjklmnpqrstvwxyz]{4})\w*",
            s, re.IGNORECASE)]
```

Izraz išče poljubno število črk (`\w*`), ki ji sledita dva samoglasnika (`[aeiou]{2}`) ali štirje soglasniki (`[bcdfghjklmnpqrstvwxyz]{4}`) in nato spet poljubno število črk (`\w*`).

Izraz je grd, pa še šumnike in druge čudne črke smo izpustili. Veliko všečnejša rešitev je

```
import re
def zaporedni(s):
    return [beseda for beseda in re.findall(r"\w+", s)
            if    re.search("[aeiou]{2}", beseda, re.IGNORECASE)
               or re.search("[^aeiou]{4}", beseda, re.IGNORECASE)]
```

Iščemo vse besede, `re.findall(r"\w+", s)`, za katere velja, da vsebujejo dva zaporedna samoglasnika ali štiri zaporedne črke, ki niso samoglasniki.

Rešitvi z regularnimi izrazi delujeta pravilno tudi, če se v stavku pojavljajo ločila, ne le besede ločene s presledki.

Za slovo od naloge pa še rešitev, primerna za vsak jezik, celo zbirnik (jezik, ki ga govori procesor). V starih časih smo namreč programirali tako, da smo sestavili *končni avtomat*:

```
def zaporedni(s):
    NI_CRKA, SAM1, SOG1, SOG2, SOG3, DODAJ = range(6)
    stanje = NI_CRKA
    zacetek = 0
    besede = []
    for i, c in enumerate(s):
        if c.lower() in "aeiou":
            if stanje == NI_CRKA:
                zacetek = i
            if stanje == SAM1:
                stanje = DODAJ
            elif stanje != DODAJ:
                stanje = SAM1
        elif c.isalpha():
            if stanje == NI_CRKA:
                zacetek = i
            if stanje == SOG1:
                stanje = SOG2
            elif stanje == SOG2:
                stanje = SOG3
            elif stanje == SOG3:
                stanje = DODAJ
            elif stanje != DODAJ:
                stanje = SOG1
        else:
            if stanje == DODAJ:
                besede.append(s[zacetek:i])
            stanje = NI_CRKA
    if stanje == DODAJ:
        besede.append(s[zacetek:])
    return besede
```

Spremenljivka stanje pove, ali je bil zadnje znak ne-črka, ali imamo doslej en samoglasnik, ali imamo enega, dva ali tri soglasnike ali pa smo za besedo, v kateri se trenutno nahajamo, že odločeni, da jo bomo dodali. Vsa funkcija je le igra prehajanj med temi stanji.

243. Migracije

Najprej moramo znati prebrati datoteko po vrsticah. Nekateri tu radi pišejo neumnosti kot `for vrstica in open(ime_datoteke).read().split("\n")`. To se lahko zalomi na sto in en način. V resnici najlepše deluje preprosti `for vrstica in open(ime_datoteke)`.

Potem je potrebno razbiti vrstice. Imena nekaterih krajev so sestavljena iz več besed, zato razbijanje po presledkih ne bo delovalo. Spomniti se moramo, da lahko metodi `split` podamo tudi argument. Vrstico najprej razbijemo glede na `":"`, nato glede na `"->"`. Torej

```
koliko, kraja = vrstica.split(":")
odkod, kam = kraja.split("->")
```

Deluje tudi obratno, le nekako bolj logično je tako.

Potem seštevamo prišleke in odšleke v dva slovarja s privzetimi vrednostmi. Na koncu vrnemo ključa, ki ustrezata največji vrednosti v vsakem slovarju.

```
def migracije(ime_datoteke):
    from collections import defaultdict

    viri = defaultdict(int)
    ponori = defaultdict(int)
    for vrstica in open(ime_datoteke):
        koliko, kraja = vrstica.split(":")
        odkod, kam = kraja.split("->")
        viri[odkod.strip()] += int(koliko)
        ponori[kam.strip()] += int(koliko)
    return naj_kljuc(viri), naj_kljuc(ponori)
```

V `return`-u kličemo funkcijo `naj_kljuc`, ki vrne ključ, ki pripada največji vrednosti. To se nam splača storiti zato, da ne bomo dvakrat programirali istega. Funkcijo moramo seveda napisati - kar pa ne bi smel biti problem.

```
def naj_kljuc(d):
    naj_k = None
    naj_v = 0
    for k, v in d.items():
        if v > naj_v:
            naj_k, naj_v = k, v
    return naj_k
```

Funkciji `naj_kljuc` se lahko sicer izognemo s čarovnijo: `return` v funkciji `migracije` spremenimo v

```
    return max(viri, key=viri.get), max(ponori, key=ponori.get)
```

244. Selitve

Pripravili si bomo slovar, katerega ključi so imena krajev, pripadajoče vrednosti pa množice prebivalcev tega kraja.

Naslednji podvig: kako priti do imen krajev v vsaki vrstici datoteke? Preprosto. Ker v vrstici ni drugih narekovajev, vrstico razbijemo glede na narekovaje. Druga in tretja reč sta iskana kraja. (Da bo koda preglednejša, vrstico razpakiramo, odvečnim spremenljivkam pa damo ime _.)

Ostalo je preprosto.

```
def selitve(zacetek, navodila, kraj):
    lokacije = {kraj: {prebivalec} for kraj, prebivalec in zacetek}
    for vrstica in open(navodila):
        _, odkod, _, kam, _ = vrstica.strip().split('"')
        lokacije[kam] |= lokacije[odkod]
        lokacije[odkod] = set()
    return lokacije[kraj]
```

Funkcija, ki smo jo napisali tule, predpostavlja, da se prebivalci nekega kraja nikoli ne bodo selili v isti kraj - Kamničani se nikoli ne selijo v Kamnik. Kaj bi se zgodilo, če bi bila v datoteki tudi taka selitev? Kako bi popravili funkcijo?

245. Navodila

Ko naletimo na števko (torej nekaj, kar ni L ali D), jo shranimo v niz, v katerega shranjujemo števke.

Ko naletimo na L ali D, to (morda) pomeni konec nekega števila. Ali je tako, prepoznamo po tem, da niz s števkami ni prazen. Če torej ni, v seznam dodamo to število. V vsakem primeru pa dodamo še L ali D in spraznimo niz s številom.

```
def navodila(zaporedje):
    stevilo = ""
    koraki = []
    for c in zaporedje:
        if c in "LD":
            if stevilo:
                koraki.append(int(stevilo))
                stevilo = ""
            koraki.append(c)
        else:
            stevilo += c
    if stevilo:
        koraki.append(int(stevilo))
    return koraki
```

Praktično vsi, ki so rešili nalogo, so jo rešili na tak način (le z malo več `if`-i, ker ste običajno ločeno obravnavali `"L"` in `"D"`, nekateri pa so komplicirali z `isdigit` in podobnimi metodami nizov).

Malo krajši način je tale: namesto da bi število računali v ločenem nizu, ga zapišemo kar v končni seznam. Če mu sledi drugo število, ga le pomnožimo z 10 in prištejemo naslednjo števko.

```
def navodilo(zaporedje):
    koraki = []
    for c in zaporedje:
        if c in "LD":
            koraki.append(c)
        elif not koraki or koraki[-1] in "LD":
            koraki.append(int(c))
        else:
            koraki[-1] = 10 * koraki[-1] + int(c)
    return koraki
```

Če naletimo na L ali D, ga dodamo v seznam. Sicer pa preverimo ali je zadnji element seznama L ali D (ali pa je ta celo prazen). V tem primeru dodamo `c`, pretvorjen v številko. Če imamo na koncu seznama že številko, pa jo pomnožimo z 10 in prištejemo `c`.

246. Poet tvoj nov Slovencem venec vije

Najprej razmislimo, kako bomo prišli do zadnje besede vsake vrstice.

```
def rime(pesmica):
    for vrstica in pesmica.split("\n"):
        zadnja_beseda = vrstica.split(" ")[-1]
        if not zadnja_beseda:
            continue
        zlog = zadnja_beseda.split("-")[-1].strip(" .,:;)?!")
```

Celotna pesem je podana kot en sam niz, zato ga razbijemo na vrstice s `split("\n")`. Vrstico razbijemo na besede (`split(" ")`) in vzamemo zadnjo besedo. Če je bila `vrstica` prazna, bo prazna tudi `zadnja_beseda`, v tem primeru vrstico preskočimo (`continue`). Sicer razbijemo besedo na zloge (`split("-")`). Vzamemo zadnji zlog; če je beseda enozložna, bomo dobili kar celo besedo. Od besede odluščimo morebitna ločila (`strip(" .,:;)?!")`).

Zdaj, ko smo pridelali zadnji zlog, nas navodila naloge vodijo za roko:

- vrstici, ki se ne rima z nobeno predhodno vrstico, priredimo novo črko; črke jemljemo po abecedi
- vrstici, ki se rima s predhodno, priredimo enako črko kot tej vrstici.

Sestavili bomo slovar, katerega ključi bodo zlogi, vrednosti pa znaki, dodeljeni zlogu. Če zloga še ni, mu dodelimo nov znak in ga dodamo v slovar.

```
def rime(pesmica):
    dodeljene = {}
    rima = ""
    for vrstica in pesmica.split("\n"):
        zadnja_beseda = vrstica.split(" ")[-1]
        if not zadnja_beseda:
            continue
        zlog = zadnja_beseda.split("-")[-1].strip(" .,:;)?!")
        if zlog not in dodeljene:
            dodeljene[zlog] = "ABCDEFGHIJKLMNOPRSTUVZ"[len(dodeljene)]
        rima += dodeljene[zlog]
    return rima
```

Če poznamo metode slovarja, lahko ključne tri vrstice zamenjamo z eno samo

```
rima += dodeljene.setdefault(zlog,"ABCDEFGHIJKLMNOPRSTUVZ"[len(dodeljene)])
```

Metoda `setdefault` bo dodala novi znak, če ga še ni, sicer pa bo vrnila obstoječega – natančno, kar potrebujemo. Znebimo se lahko tudi niza s črkami; novo črko lahko naračunamo.

```
rima += dodeljene.setdefault(zlog, chr(ord("A") + len(dodeljene))
```