

MOL vzdržuje spletno stran Pobude meščanov. Z namenom zagotavljanja čim resnejših odgovorov na pobude pešcev kolesarjev, je Oddelek za gozdarske dejavnosti in motorni promet vzpostavil hierarhijo in postopek odgovarjanja, ki je takšen:

1. Vsako prejeto pobudo zabeležijo in jo dodelijo v obravnavo naključno izbrani osebi iz Oddelka.
2. Oseba pobudo temeljito preuči in nato:
 - prepusti pobudo v obravnavo kateremu izmed podrejenih (če jih ima).
Ista oseba je lahko podrejena večim osebam!
 - Druga možnost je, da na pobudo odgovori sama, tako da naključno izbere enega izmed naslednjih štirih odgovorov:
 - Hvala za komentar.
 - Predlagana sprememba ni v načrtu.
 - Ureditev bomo reševali celostno.
 - Problem bomo preučili in ukrepali skladno s pristojnostmi.

Severnokorejski hekerji so nedavno vdrl v prostore Oddelka in jim odtujili dragoceni USB ključek z zapiski delovanja Oddelka ter javno objavili njegovo vsebino. Vsaka vrstica vsebuje

- datum, ko se je s pobudo nekaj zgodilo. Zapisan je s številom dni od vzpostavitve spletne strani,
- številko pobude,
- oznako, da je pobuda prejeta ali pa ime osebe, ki je v zvezi s pobudo na ta dan izvedla bodisi akcijo "odgovori" bodisi akcijo "prepusti".
- Akciji sledi dvopičje in nato bodisi odgovor bodisi ime osebe, ki ji je bila pobuda prepuščena.

...

```
[709] Pobuda 108: Angelca prepusti: Cilka
[710] Pobuda 124: prejem
[711] Pobuda 110: Zofka prepusti: Johanca
[715] Pobuda 125: prejem
[718] Pobuda 101: Fanči odgovori: Hvala za komentar.
[720] Pobuda 115: Zofka prepusti: Johanca
[720] Pobuda 126: prejem
[724] Pobuda 99: Johanca prepusti: Ema
```

...

Datoteka je urejena kronološko, po naraščajočih časih.

Točno obliko si oglejte v priloženi datoteki.

Ocena 6

Napiši naslednje funkcije.

`preberi_zapis(s)`

- `preberi_zapis(s)` prejme vrstico, kakršne vidite v datoteki, in vrne terko s petimi elementi (`datum`, `stevilka`, `akcija`, `kdo`, `rezultat`).
 - `datum` in `stevilka` sta števili (`int`)
 - `akcija` je "prejem", "odgovori" ali "prepusti"
 - `kdo` je ime osebe, ki je pobudo obravnavala in jo na ta dan prepustila drugemu ali pa odgovorila nanjo; v primeru, da je akcija "prejem", je `kdo` enak `None`
 - `rezultat` je ime osebe, ki ji je bila pobuda prepuščena (če je `akcija` enaka "prepusti"), odgovor (če je `akcija` "odgovori") ali `None` (če je `akcija` "prejem").

Primeri:

- `preberi_zapis("[720] Pobuda 126: prejem")` vrne (720, 126, "prejem", `None`, `None`).
- `preberi_zapis("[725] Pobuda 86: Ema odgovori: Ureditev bomo reševali celostno.")` vrne (725, 86, "odgovori", "Ema", "Ureditev bomo reševali celostno.")
- `preberi_zapis("[727] Pobuda 114: Zofka prepusti: Johanca")` vrne (727, 114, "prepusti", "Zofka", "Johanca").

Rešitev Tole je očitno vaja iz nizov in indeksov. Pravzaprav nekoliko sitna vaja iz nizov in indeksov. :) Ker oboje poznamo že dolgo, jo vzemimo predvsem kot vajo iz dobre organizacije funkcije - takšne, pri kateri se ne bomo izgubili v tem, kje smo, kaj smo že razbrali iz niza in kje je shranjeno.

Ob pisanju rešitve (dva tedna po sestavljanju naloge) sem razmišljal, da je to pravzaprav odlična naloga. Dobrega programerja ne spoznaš po tem, kako zvito zna obračati slovarje, temveč po tem, ali bo pravilno in, predvsem, pregledno rešil zoprno vsakdanjo nalogo. Če ti tule napiše packarijo, o kateri lahko rečemo le, da je videti, da deluje ... potem ga mogoče ne potrebujemo.

Ena rešitev je lahko takšna:

```
def preberi_zapis(s):
    deli = s.split(": ")
    cas, _, stevilka = deli[0].split()
    cas = int(cas[1:-1])
    stevilka = int(stevilka)

    if deli[1] == "prejem":
        return (cas, stevilka, "prejem", None, None)
    else:
        kdo, kaj = deli[1].split()
        return (cas, stevilka, kaj, kdo, deli[2])
```

Naloga je prijazna v toliko, da so deli vrstice ločeni z dvopičji - enim ali dvema. Začnimo torej s `split(':')`. Pazite: `:` in ne `:`, saj se tako mimogrede znebimo še presledka.

Nato iz prvega dela izločimo čas in številko, saj ju bomo v vsakem primeru potrebovali. To storimo kar s `split()`, pri čemer bomo vedno dobili tri stvari, vendar nas druga ne zanima, zato jo "shranimo" kar v spremenljivko z neuglednim (neopaznim) imenom `_`. Od časa odbijemo prvi in zadnji znak, oglate oklepaje in oboje pretvorimo v `int`.

Nato preverimo, ali gre z prejem. V tem primeru že vemo, kaj vrniti.

Sicer gre za odgovor ali prepuščanje. V vsakem primeru razdelimo drugi del, tisti, ki je med dvopičjema. Tam bo pisalo, kdo je naredil in kaj je naredil. Odgovor, ki ga je dal, ali osebo, ki ji je prepustil pobudo, pa imamo že v zadnjem delu.

Rešitev ni preveč grozna, moti me le preobilica indeksov. Podatke raje imenujem kot indeksiram. Morda je boljše tako:

```
def preberi_zapis(s):
    cas_stevilka, akcija, *rezultat = s.split(": ")
    cas, _, stevilka = cas_stevilka.split()
    cas = int(cas[1:-1])
    stevilka = int(stevilka)

    if akcija == "prejem":
        return (cas, stevilka, "prejem", None, None)
    else:
        kdo, kaj = akcija.split()
        return (cas, stevilka, kaj, kdo, rezultat[0])
```

Tale rešitev zahteva, da znamo razpakirati seznam z neznanim številom elementov: `s.split(": ")` vrne dve ali tri reči. Prirejanje `cas_stevilka`, `akcija`, `*rezultat = ...` bo priredil prvo reč imenu `cas_stevilka`, drugo `akcija`, vse ostale pa shranil v terko `rezultat`.

Da se ne bi kdo pritoževal, da vas učim stvari specifične za Python: ideja je popularna v različnih jezikih, samo sintaksa je tu takšna, tam drugačna. Tole je ista reč v JavaScriptu:

```
> const [a, b, ...c] = [1, 2, 3, 4, 5]
undefined
> a
1
> b
2
> c
[ 3, 4, 5 ]
```

Naprej gre tako kot prej, le namesto `deli[0]`, `deli[1]` in `deli[2]` uporabljamo

jasnejša imena. Kazi le `rezultat[0]` na koncu. Tako pač je: `rezultat` je terka z največ enim elementom. Če koga moti, lahko piše `return (cas, stevilka, kaj, kdo) + rezultat`, vendar je to slabše, saj bi ob tem morda kdo pomislil celo, da ima `rezultat` lahko več elementov.

Naloga za oceno 10 je zahtevala, da uvedemo poimenovano terko `Zapis` in označimo tipe argumentov in rezultata. Storimo tako.

Terka `Zapis` bo takšna:

```
from typing import NamedTuple
```

```
class Zapis(NamedTuple):
    cas: int
    stevilka: int
    akcija: str
    kdo: str = None
    rezultat: str = None
```

Imena in tipi so takšni, kot zahteva naloga, in zadnji dve lastnosti imata privzeto vrednost `None`. Zdaj jo uporabimo, pa še tip argumenta in rezultata označimo.

```
def preberi_zapis(s: str) -> Zapis:
    cas_stevilka, akcija, *rezultat = s.split(": ")
    cas, _, stevilka = cas_stevilka.split()
    cas = int(cas[1:-1])
    stevilka = int(stevilka)

    if akcija == "prejem":
        return Zapis(cas, stevilka, "prejem")
    else:
        kdo, kaj = akcija.split()
        return Zapis(cas, stevilka, kaj, kdo, rezultat[0])
```

Spremenili smo le prvo vrstico in oba `return`, pri čemer smo v prvem pobrisali `None, None`, ki ju ni potrebno podajati, saj imata privzete vrednosti.

Ne bi bilo nepričakovano, da bi se kdo tega lotil reševati z regularnimi izrazi; spoznal in vzljubil jih je v srednji šoli. Študenti jih imajo včasih preradi in jih uporabljajo tudi, kjer si z njimi samo zapletejo življenje. Tule ... no, še kar delajo.

```
import re
```

```
def preberi_zapis(s: str) -> Zapis:
    mo = re.match(r"\\[(\\d+)\\] Pobuda (\\d+): (\\w+)( (\\w+): (\\.*)?)?", s)
    cas, stevilka, ime, _, akcija, rezultat = mo.groups()
    cas = int(cas)
    stevilka = int(stevilka)
    if ime == "prejem":
```

```

        return Zapis(cas, stevilka, "prejem")
    else:
        return Zapis(cas, stevilka, akcija, ime, rezultat)

```

Vse, kar sledi prvemu (`\w+`), ki bo ujel bodisi besedo `prejem` bodisi ime osebe, je opcijska skupina (`(\dots)?`). Ker je potrebujemo, jo je v naslednji vrstici pogoltnila spremenljivka `_`.

Odvečni skupini bi se bilo mogoče izogniti,

```

mo = re.match(r"[(\d+)] Pobuda (\d+): (\w+) ?(\w+)?(?: ?(.*)?)", s)
cas, stevilka, ime, akcija, rezultat = mo.groups()

```

vendar smo s tem zapletli regularni izraz. Boljše je imeti dodatno spremenljivko v prirejanju kot zapleten regularni izraz.

Sam skupine rad poimenujem.

```

def preberi_zapis(s: str) -> Zapis:
    mo = re.match(r"[(?P<cas>\d+)] Pobuda (?P<stevilka>\d+): (?P<ime>\w+)( (?P<akcija>\w+))?"
    cas, stevilka, ime, akcija, rezultat = mo.group("cas", "stevilka", "ime", "akcija", "rezultat")
    cas = int(cas)
    stevilka = int(stevilka)
    if ime == "prejem":
        return Zapis(cas, stevilka, "prejem")
    else:
        return Zapis(cas, stevilka, akcija, ime, rezultat)

```

Tule pa moram priznati, da poimenovanje le zaplete regularni izraz, torej je boljša različica brez poimenovanja. Predvsem zato, ker tako ali tako potrebujemo vse skupine in njihov vrstni red je jasen.

`preveri_dnevnik(ime_datoteke)`

- `preberi_dnevnik(ime_datoteke)` prejme ime datoteke in vrne seznam terk, kot jih vrne funkcija `preberi_zapis`. Terke predstavljajo vse vrstice v datoteki v enakem vrstnem redu, kot so v datoteki.

Rešitev Vsaki vrstici odbijemo odvečni `\n` in jo prepustimo funkciji `preberi_zapis`, rezultat pa zložimo v seznam. Da ne dolgovezimo, to storimo z izpeljanim seznamom.

```

def preberi_dnevnik(ime_dat):
    return [preberi_zapis(s.strip("\n")) for s in open(ime_dat)]

```

Za oceno 10 odprimo datoteko z `with`, pa še tipe označimo.

```

def preberi_dnevnik(ime_dat: str) -> list[Zapis]:
    with open(ime_dat) as f:
        return [preberi_zapis(s.strip("\n")) for s in f]

```

`strni(zapis)`

- `strni(zapisi)` prejme seznam terk, kot jih vrne funkcija `preberi_dnevnik`, in vrne seznam, katerega *i*-ti element vsebuje seznam z vsemi terkami, ki se nanašajo na *i*-to pobudo. Terke morajo biti tem iz prejšnjih dveh funkcij, le brez številke pobude. Elementi seznama morajo biti urejeni po naraščajočem datumu.

Ker pobuda s številko 0 ne obstaja, je prvi element seznama enak ``None``. Predpostaviti

Če pokličemo ``strnjeni = strni(preberi_dnevnik("dnevnik.txt"))``, je ``strni[101]`` enak

...

```
[(554, 'prejem', None, None),
 (638, 'prepusti', 'Angelca', 'Fanči'),
 (718, 'odgovori', 'Fanči', 'Hvala za komentar.')]
...
```

saj je bila pobuda 101 preprosta in je njena obravnava zahtevala le dve osebi (in slabe

Rešitev Glavni izziv je bil pripraviti primerno velik seznam. Ena možnost je, da najprej ugotovimo število pobud, pripravimo seznam z ustreznim številom elementov in ga nato v zanki polnimo.

Tole bi bil primer ne preveč spretne rešitve:

```
def strni(zapisi):
    naj_st = 0
    for zapis in zapisi:
        if zapis[1] > naj_st:
            naj_st = zapis[1]

    pobude = [None]
    for _ in range(naj_st):
        pobude.append([])

    for pobuda in zapisi:
        pobude[pobuda[1]].append(pobuda[:1] + pobuda[2:])
    return pobude
```

V prvem delu ugotovimo največjo številko pobude.

V drugem pripravimo seznam: vsebuje `None` in nato toliko praznih seznamov, kolikor je pobud.

Nato gremo čez pobude in v seznam z indeksom `pobuda[1]` (številka pobude) dodamo vse elemente terke, ki opisuje pobudo, razen prvega.

Malenkost spretnejši zamenja celotni prvi del s klicem `max`:

```
naj_st = max(zapis[1] for zapis in zapisi)
```

Opogumljen z rezultatom se morda nameri poenostaviti še sestavljanje seznamov:

```
pobude = [None] + [[]] * naj_st
```

Ko to ne deluje, se, upamo, spomni, kaj naredi `[[]] * naj_st`: pripravi seznam, v katerem se `naj_st`-krat ponovi *en in isti* prazen seznam.

Seznam, ki vsebuje `None` in potem kup praznih seznamov, se sestavi z

```
pobude = [None] + [[] for _ in range(naj_st)]
```

Glede na to, da tako ali tako potrebujemo zanko za dodajanje seznamov, ne bo škode, če jih dodajamo sproti, po potrebi:

```
def strni(zapisi):
    pobude = [None]
    for pobuda in zapisi:
        cas, stevilka, ostalo = pobuda[0], pobuda[1], pobuda[2:]
        while stevilka >= len(pobude):
            pobude.append([])
        pobude[stevilka].append((cas, ) + ostalo)
    return pobude
```

Elemente pobude poimenujemo (`cas`, `stevilka`, `ostalo`). Če (oziroma: dokler) je seznam prekratek, ga dopolnimo. Nato pod ustrezno številko dodamo čas in ostale elemente pobude.

Nekoliko spretnejši obvlada razpakiranje in pakiranje terk ter naredi tako:

```
def strni(zapisi):
    pobude = [None]
    for cas, stevilka, *ostalo in zapisi:
        while stevilka >= len(pobude):
            pobude.append([])
        pobude[stevilka].append((cas, *ostalo))
    return pobude
```

casi(zapisi)

- `casi(zapisi)` prejme seznam terk, kot jih vrne funkcija `preberi_dnevnik`, in vrne slovar, katerega ključi so številke pobud, vrednosti pa čas (v dnevih), ki je minil od prejema pobude do odgovora nanjo.

Rešitev Bistvo te naloge je, da se spomnimo uporabiti funkcijo `strni`, ki nam že lepo združi zapise, ki se nanašajo na isto pobudo.

```
def casi(zapisi):
    casi_pobud = {}
    for stevilka, zapisi in enumerate(strni(zapisi)[1:], start=1):
```

```

    casi_pobud[stevilka] = zapisi[-1][0] - zapisi[0][0]
    return casi_pobud

```

Edini detajl je, da po `strni` nimamo več številčk pobud oziroma se te skrivajo v indeksih elementov, torej bomo uporabili `enumerate`. Ker ničta pobuda ni pobuda, preskočimo prvi zapis (`[1:]`) in štejemo od 1 (`start=1`). Kdor tega ne zna, pač piše

```

def casi(zapisi):
    casi_pobud = {}
    for stevilka, zapisi in enumerate(strni(zapisi)):
        if stevilka > 0:
            casi_pobud[stevilka] = zapisi[-1][0] - zapisi[0][0]
    return casi_pobud

```

V vsakem primeru nato pogledamo zadnji in prvi zapis, ki se nanaša na to pobudo (`zapisi[-1]` in `zapisi[0]`) ter odštejemo njuna časa.

Spretnější naredi tako:

```

def casi(zapisi):
    strnjeni = strni(zapisi)[1:]
    return {stevilka: zapisi[-1][0] - zapisi[0][0]
            for stevilka, zapisi in enumerate(strnjeni, start=1)}

```

Če bi uporabljal poimenovane terke, bi lahko pisal tudi

```

def casi(zapisi):
    strnjeni = strni(zapisi)[1:]
    return {stevilka: zapisi[-1].cas - zapisi[0].cas
            for stevilka, zapisi in enumerate(strnjeni, start=1)}

```

Žal se nisem spomnil, da bi sestavil teste tako, da bi v primeru, da je študent definiral `Zapis` vedno uporabljali `Zapis` namesto nepoimenovanih terk, zato ta rešitev ne preneha testov, čeprav je seveda pravilna in boljša.

Ocena 7

`hitro(zapisi, meja)`

Za to nalogo se je potrebno le spomniti, da imamo funkcijo `casi`, ki vrne čase. Pravzaprav vrne slovar s številčkami pobud in časi, mi pa potrebujemo le čase, torej `.values()`.

```

def hitro(zapisi, meja):
    hitri = 0
    for cas in casi(zapisi).values():
        if cas <= meja:
            hitri += 1
    return hitri

```


Preprosteje je s `sum`: spomnimo se, da je `True` (vsaj v Pythonu) isto kot 1.

```
def hitro(zapisi, meja):  
    return sum(cas <= meja for cas in casi(zapisi).values())
```

Za oceno 10 pa še označimo tipe:

```
def hitro(zapisi: list[Zapis], meja: int) -> int:  
    return sum(cas <= meja for cas in casi(zapisi).values())
```

najtezji_primer(zapisi)

- `najtezji_primer(zapisi)` vrne številko pobude, ki je bila obravnavana najdlje. Če je takšnih več, vrne poljubno med njimi.

Rešitev Tole je naloga iz začetkov našega predmeta.

```
def najtezji_primer(zapisi):  
    naj_prim = naj_cas = None  
    for primer, cas in casi(zapisi).items():  
        if naj_cas is None or cas > naj_cas:  
            naj_prim, naj_cas = primer, cas  
    return naj_prim
```

Kdor razume trik za iskanje ključa, ki pripada največjemu elementu, napiše

```
def najtezji_primer(zapisi):  
    c = casi(zapisi)  
    return max(c, key=c.get)
```

Ko bomo prišli do naloge za oceno 10, bomo rešili še krajše:

```
@s_casi  
def najtezji_primer(ca):  
    return max(ca, key=ca.get)
```

podrejeni(zapisi)

- `podrejeni(zapisi)` vrne slovar, katerega ključi so imena vse oseb, ki imajo vsaj enega podrejenega, vrednosti pa množice podrejenih. Da je neka oseba podrejena drugi, sklepamo po tem, da je od nje vsaj enkrat dobila v obravnavo neko pobudo.

Naj vas ne začudi: Oddelek za gozdarske dejavnosti in motorni promet je sofisticiran, zato ima lahko vsaka oseba več nadrejenih. Nikoli pa ne pride do kroga, ko bi bila neka oseba (posredno) podrejena sama sebi.

Rešitev Ta naloga je preprosta, samo pomisliti je potrebno, kaj hoče od nas.

```
def podrejeni(zapisi: list[Zapis]) -> dict[str, set[str]]:  
    pod = defaultdict(set)
```

```

for _, akcija, kdo, kaj in zapisi:
    if akcija == "prepusti":
        pod[kdo].add(kaj)
return pod

```

Mimogrede smo že označili tudi tipe, kot zahteva ocena 10. Predvsem je zanimiv rezultat: gre za `dict`, katerega ključi so `str`, vrednosti pa `set str`-jev.

Ocena 8

`razmerje(zapisi)`

- `razmerje(zapisi)` vrne slovar, katerega ključi so imena vseh oseb, vrednosti pa delež ponudb, ki jih je ta oseba obravnavala in nanje odgovorila sama. Če bi Cilka odgovorila na 7 pobud, 13 pa prepustila drugim, bi bil njen delež 0.35 (7 / 20).

Rešitev Ni znanost: preštejemo število pobud, ki jih je dobila oseba v roke in, ločeno, število teh, na katere je odgovorila. Na koncu v nov slovar zložimo kvociente. Ker v slovarju ni nikogar, ki ne bi dobil v roke vsaj ene pobude (kako bi prišel vanj?!), se nam tu ni bati deljenja z 0.

```

def razmerje(zapisi):
    resil = defaultdict(int)
    vseh = defaultdict(int)
    for zapis in zapisi:
        _, _, akcija, kdo, _ = zapis
        if akcija == "odgovori":
            resil[kdo] += 1
        vseh[kdo] += 1
    del vseh[None]
    return {ime: resil[ime] / n for ime, n in vseh.items()}

```

Zanimivo je, kako smo se rešili zapisov, ki se nanašajo na prejem pobude. Tam je `kdo` enak `None`. Namesto da bi to filtrirali znotraj zanke, mirno štejemo `None` kot zaposlenega na MOL in se ga znebimo šele po zanki, s preprostim `del vseh[None]`.

`natancnost(zapisi)`

- `natancnost(zapisi)` vrne slovar, katerega ključi so imena oseb, vrednosti pa povprečni čas, ki ga je oseba porabila od prejema pobude do odgovora oz. prepustitve podrejenemu.

Za pobudo 101 obstajajo naslednji zapisi:

```

[554] Pobuda 101: prejem
[638] Pobuda 101: Angelca prepusti: Fanči
[718] Pobuda 101: Fanči odgovori: Hvala za komentar.

```

To pomeni, da jo je Angelca obravnavala $638 - 554 = 84$ dni, Fanči pa $718 - 638 = 80$ dni.

Rešitev Tu moramo pomisliti, v kakšni obliki potrebujemo podatke. Vedeti želimo, koliko časa je minilo med tem, ko je oseba dobila pobudo v obravnavo in ukrepala v zvezi z njo. Da lahko to učinkovito izračunamo, bomo nujno potrebovali strnjene pobude.

```
def natancnost(zapisi):
    casi_resevanja = defaultdict(list)
    for pobuda in strni(zapisi)[1:]:
        for (cas1, *_), (cas2, _, kdo, _) in pairwise(pobuda):
            casi_resevanja[kdo].append(cas2 - cas1)
    return {ime: sum(care) / len(care) for ime, care in casi_resevanja.items()}
```

Tule je `pobuda` seznam vseh zapisov, povezanih s to pobudo. S `pairwise(pobuda)` nabereмо zaporedne pare zapisov. V obeh nas zanima čas (`cas1`, `cas2`). V drugi nas zanima še, kdo je oseba, ki je ob `cas2` nekaj naredila v zvezi s to pobudo. Vse ostale podatke pomečemo v `_`.

Slovar `casi_resevanja` bo za vsako osebo (ključ) hranil seznam časov, ki jih je ta oseba porabila za reševanja pobud. V zanki torej dodamo v `casi_resevanja[kdo]` čas, ki ga je porabila za to pobudo.

Na koncu sestavimo, kar zahteva naloga: slovar, katerega ključi so imena oseb, vrednosti pa pripadajoči povprečni `casi_resevanja`, `sum(care) / len(care)`.

kvaliteta(zapisi)

- `kvaliteta(zapisi)` vrne seznam oseb, urejen glede povprečni čas, ki ga porabijo za obravnavo, **začenši s tistimi, ki so najvestnejši in za obravnavo porabijo največ. Če ima več oseb enak čas, naj bodo urejene po abecedi.**

Rešitev Težava te naloge je v mastnem tisku. Čase želimo urediti padajoče, imena naraščajoče (po abecedi).

Osnovna rešitev je, da vse osebe razmečemo po "predalih" - slovarjih, katerega ključi so časi reševanja, vrednosti pa seznam oseb, ki so porabile takšen čas. Nato gremo po padajočih vrednostih ključev in v seznam, ki ga bomo vrnili, dodajamo urejene seznime oseb v teh predalih. Uh, tole bo preprosteje povedati v Pythonu. :)

```
def kvaliteta(zapisi):
    predali = defaultdict(list)
    for kdo, nat in natancnost(zapisi).items():
        predali[nat].append(kdo)

    kvalitete = []
```

```

for nat in sorted(predali, reverse=True):
    kvalitete.extend(sorted(predali[nat]))
return kvalitete

```

Krajša rešitev uporabi trik: sestavi seznam terk `(-natancnost, ime)`. `-natancnost` bo negativna vrednost časa, ki ga v povprečju porabi oseba. Nato ta seznam uredimo. Tisti, ki so bolj natančni, bodo imeli bolj negativno vrednost, zato bodo v seznamu prej. Te z enako natančnostjo bo uredil po drugem elementu seznama, imenu.

```

def kvaliteta(zapisi):
    kvalitete = [(-nat, ime) for ime, nat in natancnost(zapisi).items()]
    kvalitete.sort()
    return [ime for _, ime in kvalitete]

```

Kdor želi izpasti zviti, piše

```

def kvaliteta(zapisi):
    return [ime
            for _, ime in sorted((-nat, ime)
                                for ime, nat in natancnost(zapisi).items())]

```

Kdor se želi izogniti dodatnemu seznamu, uporabi `lambda`.

```

def kvaliteta(zapisi):
    return [k for k, _ in sorted(
        natancnost(zapisi).items(),
        key=lambda it: (-it[1], it[0]))]

```

Ocena 9

`neustvarjalni(zapisi)`

- `neustvarjalni(zapisi)`: notranja revizija je pokazala, da nekateri svojega dela ne jemljejo resno in na vse pobude odgovorijo enako, namesto da bi izžrebali naključni odgovor. Funkcija `neustvarjalni` naj vrne slovar, katerega ključi so imena teh lenuhov, vrednosti pa njihov (edini) odgovor na pobude.

Rešitev Gremo čez zapise in pripravimo slovar, katerega ključi so imena oseba, vrednosti pa množice vseh odgovorov, ki jih je dajala ta oseba. (V "množice" je bistvo te naloge: spomniti se moramo, da je to najpreprostejši način za pridobivanje unikatnih elementov. Pišoč ta komentar mi je prišlo na misel, da bi lahko tole domačo nalogo uporabili tudi pri predmetu s področja podatkovnih baz. :)

Nato sestavimo slovar, katerega ključi so imena oseb in njihovi odgovori ... za vse osebe, katerih število različnih odgovorov je enako 1.

```
def neustvarjalni(zapisi):
    odgovori = defaultdict(set)
    for _, _, akcija, kdo, kaj in zapisi:
        if akcija == "odgovori":
            odgovori[kdo].add(kaj)

    return {ime: odg.pop() for ime, odg in odgovori.items() if len(odg) == 1}
```

Zanimivost je `odg.pop()`: to je najpreprostejši način, da iz množice izbezamo njen edini element - če nam je seveda vseeno, da množico pri tem "pokvarimo". In tu nam je.

`pot_med(hierarhija, oseba1, oseba2)`

- `pot_med(zapisi, oseba1, oseba2)`: če je `oseba2` posredno ali neposredno podrejena osebi `oseba1`, funkcija vrne seznam oseb na poti od `oseba1` do `oseba2`. Če ni tako, vrne `None`.

Če bi na MOL delala Adamova rodbina, bi klic `pot_med(rodbina, "Adam", "Herman")` vrnil `["Adam", "Daniel", "Hans", "Herman"]`.

Rešitev Jej, končno naloga iz rekurzije!

```
from typing import Optional
```

```
def pot_med(hierarhija: dict[str, set[str]], oseba1: str, oseba2: str) -> Optional[list[str]]:
    if oseba1 == oseba2:
        return [oseba1]
    for podrejeni in hierarhija.get(oseba1, []):
        pot = pot_med(hierarhija, podrejeni, oseba2)
        if pot:
            return [oseba1] + pot
    return None
```

Če gre za eno in isto osebo (`oseba1 == oseba2`), je pot pač `[oseba1]`.

Sicer gremo po vseh podrejenih osebah in se pozanimamo za pot od njih do osebe2. Če ta obstaja, je pot od osebe1 do osebe2 enaka tej poti, le še oseba1 pripnemo na začetek. Če ni, nadaljujemo z naslednjim podrejenim. Podrejene dobimo z `hierarhija.get(oseba1, [])`: oseba morda nima podrejenih; v tem primeru je ni v slovarju `hierarhija`, zato priskrbimo privzeti seznam podrejenih - prazen seznam, `[]`.

Če iskanje prek podrejenih ne obrodi sadov, vrnemo `None`.

Naloga za oceno 10 je zahtevala označevanje tipov, zato da ste morali enkrat uporabiti še `Optional` (ali `Union`, kar želite). Funkcija namreč lahko vrne seznam nizov, `list[str]` ali pa nič.

`nivoji(hierarhija, oseba)`

- `nivoji(hierarhija, oseba)` vrne slovar, katerega ključi so vse osebe, ki so podrejene podani osebi, pripadajoče vrednosti pa število korakov od podane osebe do te osebe. **Če je do te osebe možnih več poti, mora ključ predstavljati dolžino najkrajše poti.**

```
hierarhija =  
    a: {b, c},  
    b: {c, d, e},  
    d: {e},  
}
```

Klic `nivoji(hierarhija, "a")` vrne `{"b": 1, "c": 1, "d": 2, "e": 2}`. Od a do e obstajata dve poti `a -> b -> d -> e` in `a -> b -> e`. Funkcija vrne 2, ker je to dolžina krajše poti.

Rešitev Jej, še ena naloga iz rekurzije. Mogoče.

Vse podrejene vprašamo za slovarje nivojev pod njimi. Te slovarje združujemo: ko pridobimo slovar vsakega podrejenega, za vsako ime preverimo, ali ga še nimamo v slovarju ali pa ga imamo, vendar z višjim nivojem, kot ga ponuja ta podrejeni. Če je kaj od tega res, v slovar dodamo (oziroma spremenimo) nivo za to osebo. Enak je nivoju, ki ga ima do te osebe podrejeni in še 1 zraven, saj je podrejeni za en nivo nižje.

```
def nivoji(hierarhija: dict[str, set[str]], oseba: str) -> dict[str, int]:  
    nivo = {oseba: 0}  
    for podrejeni in hierarhija.get(oseba, []):  
        for ime, n in nivoji(hierarhija, podrejeni).items():  
            if ime not in nivo or nivo[ime] > n + 1:  
                nivo[ime] = n + 1  
    return nivo
```

Slabost te rešitve je, da je časovno potratna: če je do določene osebe možno priti na več načinov, bomo funkcijo za to osebo poklicali večkrat. In vemo, kam lahko to pripelje. Temu se lahko izognemo z memoizacijo ali pa tako, da namesto rekurzivne rešitve napišemo rešitev z *iskanjem v širino*:

```
def nivoji(hierarhija: dict[str, set[str]], oseba: str) -> dict[str, int]:  
    nivo = {}  
    preveriti = [(oseba, 0)]  
    for ime, n in preveriti:  
        if ime not in nivo:  
            nivo[ime] = n  
            preveriti += ((pod, n + 1) for pod in hierarhija.get(ime, []))  
    return nivo
```

Ta postopek boste podrobno spoznavali pri predmetih s področja algoritmov.

Tule povejmo le, da hierarhijo pregleduje po nivojih: osebe, ki so bližje podani oseba, pridejo na vrsto prej, zato nam ni potrebno preverjati, ali se oseba s tem imenom že nahaja v slovarju, vendar z večjim nivojem. Še več, če se oseba že nahaja v slovarju, vemo, da smo nanjo naleteli že prej, nivo je bil manjši, zato smemo ignorirati to osebo kot tudi vse njene podrejene, saj so bili v seznam preveriti dodane že prej.

Ocena 10

Poglej ustrezne zapiske oz. videe v Božični priboljški.

- Poskrbi, da bo funkcija `preberi_dnevnik` datoteko odprla v bloku `with`.
- Za shranjevanje zapisov namesto `terk` uporabljaj poimenovane terke (`NamedTuple`) s polji `datum`, `stevilka`, `akcija`, `kdo`, `rezultat`. Ime podatkovnega tipa naj bo `Zapis`. Zadnja dva elementa naj imata privzeto vrednost `None`. (Ker je poimenovana terka še vedno terka, bo dovolj uvesti razred in spremeniti eno samo funkcijo.)
- Nekatero funkcijo, ki prejmejo argument `zapisi`, bi v resnici lahko prejele kar vrača funkcija `casi`. Tvoja funkcija `najtezji_primer` se skoraj gotovo začne z

```
def najtezji_primer(zapisi):
    casi_zapisov = casi(zapisi)
    ... # in potem delaš s casi_zapisov
```

Pripravi dekorator `s_casi`, ki ti bo omogočil to funkcijo spremeniti v

```
@s_casi
def najtezji_primer(casi_zapisov):
    ... # in potem delaš s casi_zapisov
```

Testi jo bodo seveda še vedno klicali s seznamom `terk`, dekorator pa poskrbi, da bo dobila argument v pravi obliki.

Dekorator ustrezno uporabi v funkciji `najtezji_primer` in drugih, kjer je to primerno.

- Označi tipe argumentov in rezultatov funkcij `hitro`, `podrejeni`, `pot_med` in `nivoji`.

Rešitev

Vse smo že postorili, le dekorator nam še manjka. Preprost je.

```
def s_casi(f):
    def ovita(zapisi):
        return f(casi(zapisi))
    return ovita
```

Če bi želeli biti natančni, bi pisali

```
from functools import wraps
```

```
def s_casi(f):  
    @wraps(f)  
    def ovita(zapisi):  
        return f(casi(zapisi))  
    return ovita
```

`wraps` je dekorator, ki poskrbi, da je `ovita` videti tako, kot je bila podana funkcija `f`: njeno ime postane `ovita`, popravi tudi imena njenih argumentov in oznake tipov, da se le-ti ne izgubijo ob dekoriranju.