4_resitev

January 28, 2024

0.1 1. Točni vlaki Slovenskih železnic (hipotetični, miselni eksperiment)

Napiši funkcijo `tocni(redi, dejanski)`, ki prejma dva seznama seznamov. Vsak element se nanaša na en vlak; istoležni elementi se nanašajo na iste vlake. Prvi seznam vsebuje predvidene čase prihoda na postaje, drugi pa dejanske prihode. Časi so podani v minutah od polnoči. Funkcija naj vrne število vlakov, ki niso na nobeni postaji zamujali več kot dvajset minut (kar uporabniki Slovenskih železnic štejemo za praktično točen prihod). Klic

```
tocni([[570, 590, 616, 620], [1200, 1500], [800, 900, 1000], [700, 800]],
      [[570, 611, 622, 630], [1200, 1510], [810, 910, 1000], [800, 900]]))
```

vrne 2: "točna" sta bila drugi in tretji vlak. Prvi in četrti sta vsaj enkrat zamudila več kot 20 minut.

Namig: če želiš, si lahko napišeš tudi pomožno funkcijo tocen(red, dejansko), ki prejme seznam za en sam vlak in vrne True, če je bil točen in False, če je kje zamudil za več kot 20 minut. (Če je ne napišeš, ignoriraj teste zanjo.)

0.1.1 Rešitev

Ubogajmo, napišimo funkcijo `tocen`. Gremo čez vse pare predvidenih in dejanskih časov; če naletimo na zamudo, vrnemo `False`, sicer, na koncu, po zanki, vrnemo `True`.

[1]:
```
def tocen(red, dejansko):
    for cas_red, cas_dejansko in zip(red, dejansko):
        if cas_dejansko - cas_red > 20:
            return False
    return True
```

Naprej je preprosto: gremo čez vse vlake in za vsakega, ki je bil točen, prištejemo 1.

[2]:
```
def tocni(red, dejansko):
    tocnih = 0
    for r, d in zip(red, dejansko):
        if tocen(r, d):
            tocnih += 1
    return tocnih
```

Če znamo uporabljati izpeljane sezname (in če vemo, da je `False` isto kot `0` in `True` isto kot `1`), sta funkciji lahko tudi takšni:

[3]:
```
def tocen(red, dejansko):
    return all(d - r <= 20 for r, d in zip(red, dejansko))

def tocni(redi, dejanski):
    return sum(tocen(red, dejansko) for red, dejansko in zip(redi, dejanski))
```

Brez dodatne funkcije je malo bolj zoprno. Ena od za silo elegantnih rešitev je `else` po `for`:

[4]:
```
def tocni(red, dejansko):
    tocnih = 0
    for red1, dejansko1 in zip(red, dejansko):
        for predviden, prisel in zip(red1, dejansko1):
            if prisel > predviden + 20:
                break
        else:
            tocnih += 1
    return tocnih
```

Obstajajo pa seveda tudi druge.

Kdor je prelen za pisanje dodatne funkcije, zna pa uporabljati izpeljane sezname, pa zbaše vse skupaj v

[5]:
```
def tocni(red, dejansko):
    return sum(
        all(prisel <= predviden + 20 for prisel, predviden in zip(pri, pre))
        for pri, pre in zip(red, dejansko)
    )
```

0.2 2. Enake linije

Dve liniji sta "enaki", če vsebujeta iste postaje **ali** pa imata isto začetno in končno postajo ter ena od njiju vsebuje vse postaje, ki jih vsebuje druga; v tem primeru je druga hitrejša različica prve. Noben vlak ne ustavi večkrat na isti postaji.

Napiši funkcijo `enaki(linija1, linija2)`, ki prejme seznama imen postaj in vrne `True`, če sta liniji enaki in `False`, če ne. Funkcija naj ignorira vrstni red (razen prve in zadnje postaje, ki morata biti enaki): liniji Ljubljana - Sevnica - Trebnje - Novo mesto in Ljubljana - Trebnje - Sevnica - Novo mesto sta enaki.

- Liniji Ljubljana - Kresnice - Jevnica - Litija - Zagorje - Zidani most in Ljubljana - Jevnica - Zidani most sta enaki.
- Liniji Ljubljana - Kresnice - Zidani Most in Ljubljana - Zagorje - Zidani most nista enaki, ker vsaka vsebuje kakšno postajo, ki je druga nima.
- Liniji Kresnice - Zagorje in Ljubljana - Kresnice - Zagorje nimata enake začetne postaje.

0.2.1 Rešitev

Študent naj bi razmislil, kaj pomeni, da ima ena linija vse postaje, ki jih ima druga: gre za podmnožice. Seznama torej pretvorimo v množice in preverimo ali je ena podmnožica druge oziroma

obratno. Poleg tega pa preverimo še prvo in zadnjo postajo, torej:

[6]:
```
def enaki(linija1, linija2):
    return linija1[0] == linija2[0] \
        and linija1[-1] == linija2[-1] \
        and (set(linija1) <= set(linija2)
             or set(linija2) <= set(linija1))
```

Ne spreglejte oklepajev. Imamo tri stvari: prvi postaji sta enaki IN zadnji sta enaki IN (prva je podmnožica druge ali pa je druga podmnožica prve). Brez teh oklepajev rešitev ni pravilna, ker ima `and` prednost pred `or`.

0.3 3. Čisto enake linije

Napiši funkcijo `enaki_red(linija1, linija2)`, ki poleg tega zahteva, da se postaje pojavljajo v enakem vrstnem redu.

- Liniji Ljubljana - Zalog - Laze - Kresnice - Jevnica - Litija - Zagorje in Ljubljana - Jevnica - Kresnice - Laze - Zagorje bi bili v prejšnji nalogi enaki, tu pa nista, ker imata različen vrstni red postaj (drugi se iz Jevnice vrača v Laze).

Namig 1: nalogi sta dovolj različni, da ti ena funkcija najbrž ne bo praktično nič pomagala pri drugi.

Namig 2: če liniji nista enako dolgi, je hitrejša tista, ki je krajša. Na začetku funkcije se ti morda splača napisati

```
if len(linija1) < len(linija2):
    linija1, linija2 = linija2, linija1
```

0.3.1 Rešitev

Tole je pa "ta težja" naloga izpita. Najprej preverimo prvo in zadnjo postajo. Potem poskrbimo, da je `linija1` tista, ki gre čez več postaj (kot pravi namig).

Zdaj si predstavljajte, da greste s prstoma čez seznam postaj. Z enim čez počasni vlak in ga primerjate s prstom, ki kaže na hitri vlak. Kadar počasni vlak pride na postajo, na kateri ustavlja tudi hitri, premaknete prst na hitrem vlaku za eno postajo naprej. Ko je "počasni" prst na koncu, mora biti na koncu tudi hitri vlak.

Z zanko gremo čez počasnejši vlak, kje je prst na drugem, pa bo shranjeno v indeksu (`i2`, v spodnjem programu). Indeks se poveča, ko se imeni postaj ujemata.

[7]:
```
def enaki_red(linija1, linija2):
    if linija1[0] != linija2[0] or linija1[-1] != linija2[-1]:
        return False
    i2 = 0
    if len(linija1) < len(linija2):
        linija1, linija2 = linija2, linija1
    for postaja1 in linija1:
        if i2 < len(linija2) and linija2[i2] == postaja1:
            i2 += 1
```

```
    return i2 == len(linija2)
```

0.4 4. Kopičenje zamud

Napiši funkcijo `zamujenih(vlak, cakajoci)`, ki prejme niz s kodo vlaka, ki zamuja in slovar, katerega ključi so kode vlakov, vrednosti pa množice vseh vlakov, ki morajo čakati, kadar ta vlak zamuja. Funkcija naj vrne število vseh zamujenih vlakov. Pri tem upoštevaj, da bodo zamujali tudi vlaki, ki čakajo zamujene vlake, ki čakajo zamujene vlake …

V primeru iz testov je tako: če zamuja LP3682, bosta zaradi tega zamujala tudi LP8524 in IC021. Ker zamuja LP8524 bodo morali čakati tudi EN123, IC521 in LP6316, zaradi IC021 pa LP222 in IC204. Zaradi EN123 ne čaka nihče, zaradi IC521 pa … in tako naprej. Skupno zaradi LP3682 zamudi 14 vlakov.

Vlaki se nikoli ne čakajo v krogu: če A čaka B in B čaka C, potem ne B ne C ne čakata na A.

0.4.1 Rešitev

Vrniti morate velikost "rodbine vlakov" - kar je prva naloga iz rekurzije, ki smo jo delali na predavanjih…

[8]:
```
def zamujenih(vlak, cakajoci):
    zamud = 1
    for vlak1 in cakajoci[vlak]:
        zamud += zamujenih(vlak1, cakajoci)
    return zamud
```

Malo hitreje pa je tako:

[9]:
```
def zamujenih(vlak, cakajoci):
    return 1 + sum(zamujenih(vlak1, cakajoci) for vlak1 in cakajoci[vlak])
```

Kdor ve kaj več, je opazil, da ni tako preprosto: lahko bi se zgodil, da bi na A čakala B in C, potem pa bi na oba, torej na B in na C, čakal nek vlak D. Tale program bi D štel dvakrat. To bi lahko rešili tako, da bi namesto preštevanja našteli vsa imena v rodbini in pogledali, koliko jih je.

0.5 5. Robotski sprevodnik

Ker se potniki jezijo na sprevodnike namesto na šefe SŽ, bodo uvedli robotske sprevodnike. Napiši razred `Sprevodnik`:

- konstruktor prejme začetno postajo in cenik v obliki slovarja, katerega ključi so pari imen postaj, vrednosti pa cena prevoza med tem parom;
- `postaja(ime)` se bo poklicala, ko vlak pride na postajo s podanim imenom;
- `potnik(kam)` proda vstopivšemu potniku vozovnico od trenutnega kraja do kraja s podanim imenom;
- `blagajna()` vrne trenutni izkupiček od prodanih vozovnic;
- `potnikov()` vrne trenutno število potnikov na vlaku.

Namig: Sprevodnik mora poznati trenutno postajo in število potnikov, ki izstopijo na posamezni od prihodnjih postaj.

0.5.1 Rešitev

Kot je pri nalogah iz objektnega programiranja običajno, moramo le razmisliti, kaj si bo moral zapomniti sprevodnik in kako to shraniti. Potem metode le spreminjajo te podake.

Zapomniti si bo moral - cenik (to bo `self.cenik`), - koliko ima v blagajni (to bi lahko bilo `self.blagajna`, vendar se atribut ne more imenovati kot metoda, torej recimo temu `self.zasluzak`), - trenutno postajo, kot prijazno pove namig (`self.trenutna_postaja`) in - število potnikov, ki bodo izstopili na posamezni postaji (`self.potniki`). To bomo shranjevali v slovarju, katerega ključi bodo potniki, vrednosti pa število potnikov, ki tam izstopijo.

[10]:
```
from collections import defaultdict


class Sprevodnik:
    def __init__(self, zacetna, cenik):
        self.cenik = cenik
        self.trenutna_postaja = zacetna
        self.zasluzek = 0
        self.potniki = defaultdict(int)

    def postaja(self, postaja):
        self.trenutna_postaja = postaja
        if self.trenutna_postaja in self.potniki:
            del self.potniki[self.trenutna_postaja]

    def potnik(self, kam):
        self.zasluzek += self.cenik[self.trenutna_postaja, kam]
        self.potniki[kam] += 1

    def blagajna(self):
        return self.zasluzek

    def potnikov(self):
        return sum(self.potniki.values())
```

Kaj dela konstruktor, je očitno.

- `postaja`: ko pridemo na postajo, si zapomnimo, da je to trenutna postaja. Če na taj postaji kdo izstopi, preprosto pobrišemo ta ključ iz slovarja.
- `potnik`: ko potnik vstopi in pove, kam gre, povečamo zaslužek in povečamo število potnikov, ki izstopijo na njegovi ciljni postaji.
- `blagajna` vrne zaslužek, ki smo si ga shranili v `self.zasluzek`.
- `potnikov` vrne število potnikov, ki bodo izstopili na kateri od prihodnjih postaj. Natančno toliko jih je namreč na vlaku.