resitev

January 28, 2024

Naloga vsebuje teste. Naloga je rešena samo, če prestane vse teste. Poglej navodila za poganjanje testov.

MOL je poslal zemljevid ovir na kolesarski poti. Zemljevid je shranjen kot seznam nizov, ki predstavljajo "vrstice": `#` predstavlja oviro, `.` pa prosto pot. Tipičen primer korektno zašikanirane kolesarske steze je

```
zemljevid = [
    "......",
    "..##..",
    ".##.#.",
    "...###",
    "###.##",
]
```

Napiši naslednje funkcije.

0.1 1. Dolžina ovir

`dolzina_ovir(vrstica)` prejme eno vrstico, na primer `".##.####...##."` in vrne skupno dolžino ovir, torej, število znakov `#` (v tem primeru 8).

0.1.1 Rešitev

Pripravimo si vrstico, da bomo imeli na čem preskušati funkcije. (Domači nalogi so priloženi testi, ki jih sami kličejo in preskušajo.)

[1]:
```
vrstica = ".##.####...##."
```

Prešteti moramo število `#` v nizu.

Če ne vemo ničesar od tega, česar se še nismo učili, to storimo z zanko.

[2]:
```
def dolzina_ovir(vrstica):
    ovir = 0
    for znak in vrstica:
        if znak == "#":
            ovir += 1
    return ovir
```

[3]:
```
dolzina_ovir(vrstica)
```

[3]: 8

Če se spomnimo, da je `True` isto kot `1` in `False` isto kot `0`, lahko seštevamo kar `znak == "#"`. Ta izraz bo imel vrednost `True` ali `False` torej `0` ali `1`.

[4]:
```
def dolzina_ovir(vrstica):
    ovir = 0
    for znak in vrstica:
        ovir += znak == "#"
    return ovir
```

[5]:
```
dolzina_ovir(vrstica)
```

[5]: 8

Ali je to lepo ali ne, je stvar okusa. Dobro pa je vedeti, da bomo znali nekoč napisati

[6]:
```
def dolzin_ovir(vrstica):
    return sum(znak == "#" for znak in vrstica)
```

[7]:
```
dolzina_ovir(vrstica)
```

[7]: 8

Vse to je, po drugi strani, nesmiselno, saj nizi znajo sami povedati, kolikokrat vsebujejo določen znak. Kot se bomo naučili prav v tem tednu, lahko nalogo rešimo preprosto z

[8]:
```
def dolzina_ovir(vrstica):
    return vrstica.count("#")
```

[9]:
```
dolzina_ovir(vrstica)
```

[9]: 8

0.2 2. Število ovir

`stevilo_ovir(vrstica)` prejme vrstico in vrne število ovir v vrstici. Za gornji primer mora vrniti 3, saj imamo najprej eno dvojno šikano, nato eno četverno in potem spet eno dvojno - skupaj tri.

0.2.1 Razmislek in priprava

Ovir je toliko kot začetkov ovir. Torej toliko kot znakov `#`, ki se nahajajo neposredno za znakom `.` ali pa na začetku niza.

Spet si pripravimo vrstico - tokrat takšno, ki bo imela oviro tudi čisto na začetku in na koncu, saj bosta prav tidve povzročali sitnosti.

[10]:
```
vrstica = "##.##.####...#....#"
```

Tu imamo torej `5` ovir.

Očitno spet potrebujemo zanko, s katero bomo pregledali vse znake v nizu `vrstica`. Vendar bomo poleg trenutnega znaka potrebovali tudi prejšnjega.

0.2.2 Grda rešitev: ne počnite tega ...

Začnimo z najgršo rešitvijo: zanko prek indeksov.

[11]:
```
def stevilo_ovir(vrstica):
    ovir = 0
    for i in range(len(vrstica)):
        if vrstica[i] == "#" and (i == 0 or vrstica[i - 1] == "."):
            ovir += 1
    return ovir
```

[12]:
```
stevilo_ovir(vrstica)
```

[12]:
```
5
```

Ta rešitev deluje, to pa je tudi vse, kar lahko dobrega povemo o njej.

No, mnogi jo imajo za dobro, ker je enaka rešitvi, ki bi jo napisali v kakem drugem jeziku, ki so se ga učili od prej in ki temelji na tem, da na veliko indeksiramo levo in desno, naprej in nazaj. S tem v principu ni nič narobe, to včasih počnemo tudi v Pythonu. Python pa omogoča tudi drugačen način razmišljanja pri progrmairanju, in ta včasih vodi do lepših, jasnejših, preprostejših rešitev. Poglejmo.

0.2.3 Rutinska rešitev v Pythonu: `zip` čez pare

Da dobimo pare zaporednih znakov, uporabimo `zip(vrstica, vrstica[1:])`.

[13]:
```
def stevilo_ovir(vrstica):
    ovir = 0
    for prej, zdaj in zip(vrstica, vrstica[1:]):
        if prej == "." and zdaj == "#":
            ovir += 1
    return ovir
```

[14]:
```
stevilo_ovir(vrstica)
```

[14]:
```
4
```

Funkcija vrne `4`, ker smo spregledali prvo oviro. Seveda, pred njo ni pike. Ali, z vidika gornjega programa, prvi znak niza `vrstica`, v našem zoprnem primeru ravno `#`, se nikoli ne pojavi v `zdaj`, temveč le v `prej`.

To bo preprosto rešiti: ker vemo, da bomo oviro na začetku vedno spregledali, prištejmo 1, če je prvi znak `#`.

[15]:
```
def stevilo_ovir(vrstica):
    ovir = 0
```

```
    for prej, zdaj in zip(vrstica, vrstica[1:]):
        if prej == "." and zdaj == "#":
            ovir += 1
    if vrstica[0] == "#":
        ovir += 1
    return ovir
```

[16]:
```
stevilo_ovir(vrstica)
```

[16]: 5

Ali število ovir kar v začetku nastavimo na 0 namesto na 1.

[17]:
```
def stevilo_ovir(vrstica):
    if vrstica[0] == "#":
        ovir = 1
    else:
        ovir = 0
    for prej, zdaj in zip(vrstica, vrstica[1:]):
        if prej == "." and zdaj == "#":
            ovir += 1
    return ovir
```

[18]:
```
stevilo_ovir(vrstica)
```

[18]: 5

Lahko pa uporabimo operator `if - else` (ekvivalenkt `? :` iz drugih jezikov), če vemo zanj.

[19]:
```
def stevilo_ovir(vrstica):
    ovir = 1 if vrstica[0] == "#" else 0
    for prej, zdaj in zip(vrstica, vrstica[1:]):
        if prej == "." and zdaj == "#":
            ovir += 1
    return ovir
```

[20]:
```
stevilo_ovir(vrstica)
```

[20]: 5

Pythonov operator `if - else` je čuden, nekoliko zanemarjen (ne brez razloga). Python ga je dobil šele, ko je dopolnil že 12 let, po tem, ko so leta in leta cincali, ali sploh in kako. Kogar zanima več, lahko prebere povzetek debate, ki nudi tudi zanimiv vpogled v to, kako se oblikuje jezik, kot je Python.

Tu se ga brez težav znebimo, če se spomnimo, da je `True` enak `1` in `False` enak `0`.

[21]:
```
def stevilo_ovir(vrstica):
    ovir = vrstica[0] == "#"
    for prej, zdaj in zip(vrstica, vrstica[1:]):
        if prej == "." and zdaj == "#":
            ovir += 1
    return ovir
```

[22]:
```
stevilo_ovir(vrstica)
```

[22]: 5

Tudi enica, ki jo prištevamo znotraj zanke, je samo stvar pogoja, torej bo delovalo tudi

[23]:
```
def stevilo_ovir(vrstica):
    ovir = vrstica[0] == "#"
    for prej, zdaj in zip(vrstica, vrstica[1:]):
        ovir += prej == "." and zdaj == "#"
    return ovir
```

[24]:
```
stevilo_ovir(vrstica)
```

[24]: 5

Ker sta `prej` in `zdaj` niza, ju lahko tudi seštejemo in skrajšamo pogoj.

[25]:
```
def stevilo_ovir(vrstica):
    ovir = (vrstica[0] == "#")
    for prej, zdaj in zip(vrstica, vrstica[1:]):
        ovir += (prej + zdaj == ".#")
    return ovir
```

[26]:
```
stevilo_ovir(vrstica)
```

[26]: 5

Oklepaji, v katere sem zaprl pogoje, so nepotrebni, vendar povečajo čitljivost. Tale kombinacija `+=` in`==` je sicer videti strašljiva in malenkost nepregledna.

0.2.4 Ustvarjalnejša rešitev: tudi zip čez pare

Rešitev je sicer čisto lepa, najbrž pa nam gre na živce ločeno obravnavanje prvega znaka. Meni že.

Lepo bi bilo, če bi bil z začetku `zdaj` prvi znak. Kaj pa naj bo potem `prej`? Preprosto: pika!

[27]:
```
def stevilo_ovir(vrstica):
    ovir = 0
    for prej, po in zip("." + vrstica, vrstica):
        ovir += prej + po == ".#"
    return ovir
```

Razumemo trik? Prej smo `zip`-ali

[28]:
```
print(vrstica)
print(vrstica[1:])
```

```
##.##.####…#…#
#.##.####…#…#
```

Zdaj pa zipamo

[29]:
```
print("." + vrstica)
print(vrstica)
```

```
.##.##.####…#…#
##.##.####…#…#
```

V prejšnjem primeru se je prvi znak pojavil le v prvi vrstici, v vlogi `prej`, poparjen z drugim znakom. Zdaj pa se prvič pojav v drugi vrstici, poparjen s piko, ki jo dodamo preden.

Če zdaj primerjamo rešitev, ki sem jo ozmerjal z grdo,

```
def stevilo_ovir(vrstica):
    ovir = 0
    for i in range(0, len(vrstica)):
        if vrstica[i] == "#" and (i == 0 or vrstica[i - 1] == "."):
            ovir += 1
    return ovir
```

in tole, zadnjo

```
def stevilo_ovir(vrstica):
    ovir = 0
    for prej, po in zip("." + vrstica, vrstica):
        ovir += (prej + po == ".#")
    return ovir
```

vidimo, zakaj se splača pomisliti malo drugače, brez indeksov.

0.2.5 Rešitev zares

Če dobro pogledamo zadnjo rešitev, pa vidimo, da štejemo le pojavitve `".#"` v dopolnjenem nizu. Najkrajša rešitev naloge je torej

[30]:
```
def stevilo_ovir(vrstica):
    return ("." + vrstica).count(".#")
```

[31]:
```
stevilo_ovir(vrstica)
```

[31]: 5

0.3 3. Najširša ovira

`najsirsa_ovira(vrstica)` vrne dolžino najdaljše ovire. V gornjem primeru je to 4.

0.3.1 Rešitev

Prešteti moramo število zaporednih pojavitev `#`. Ko naletimo na `.`, postavimo števec nazaj na 0. Ob vsakem povečanju števila, preverimo, ali je večje od največjega doslej.

[32]:
```
def najsirsa_ovira(vrstica):
    dolzina = 0
    naj_dolzina = 0
    for c in vrstica:
        if c == "#":
            dolzina += 1
            if dolzina > naj_dolzina:
                naj_dolzina = dolzina
        else:
            dolzina = 0
    return naj_dolzina
```

Natančneži bi ugovarjali, da je funkcija počasna, saj bi lahko preverjanje `if dolzina > naj_dolzina` opravili, ko se ovira konča, torej znotraj `else`, pred `dolzina = 0`. To je res, sitnost pa je v tem, da se `else` ne zgodi le, ko se ovira konča, temveč vsakič, kadar ni ovire. Poleg tega bi se morali posebej poukvarjati s primerom, da se ovira dotika konca vrstice.

Spet drugi so nezadovoljni zaradi dolžine programa. Nekateri bi zlovoljno dodali celo, da tole ni nič krajše, kot če bi programirali v Cju. Kako bi torej to rešili v pravem Pythonu?

Če bi moral to rešiti zares, v praksi, bi mi najprej prišlo na misel tole.

[33]:
```
def najsirsa_ovira(vrstica):
    return max(map(len, vrstica.split(".")))
```

Ali, tole - hitrejše, kadar je pik veliko:

[34]:
```
def najsirsa_ovira(vrstica):
    return max(map(len, vrstica.replace(".", " ").split()))
```

0.4 4. Pretvori vrstico

4. `pretvori_vrstico(vrstica)` vrne seznam parov `(x0, x1)`, ki predstavljajo začetke in konce ovir. Za gornji primer vrne `[(2, 3), (5, 8), (12, 13)]`. Pazi, stolpci so oštevilčeni od 1, ne 0.

0.4.1 Rešitev

Tole je glavna naloga. Na srečo smo se ob reševanju druge naloge naučili ključnega trika: če na začetek niza prištejemo ".", bomo lažje zaznali začetek prve ovire, če so nam jo nemara podtaknili na čisti začetek vrstice. Če prištejemo piko še na konec, bomo lažje zaznali zadnjo, če je morda čisto na koncu.

Tokrat se bomo za začetek pregrešili in delali z indeksi.

[35]:
```
def pretvori_vrstico(vrstica):
    vrstica = "." + vrstica + "."
    bloki = []
    for i, znak in enumerate(vrstica):
        if znak == "#":
            if vrstica[i - 1] == ".":
                zacetek = i
            if vrstica[i + 1] == ".":
                bloki.append((zacetek, i))
    return bloki
```

`znak` bo vseboval trenutni znak, `i` pa njegov indeks.

Zanimajo nas samo znaki `#`. - Če je pred njim, na indeksu `i - 1`, pika, je to začetek ovire. Zapomnimo si ga. - Če je za njim, na indeksi `i + 1`, pika, je to konec ovire. Dodamo oviro, ki se je začela pri `zacetek` (ki smo si ga morda zapomnili v enem prejšnjih krogov zanke, morda pa ravnokar, če je bila ovira slučajno dolga le en znak).

V tej nalogi stolpce štejemo od `1`, Pythonov `enumerate` pa (privzeto) teče od `0`. To je ravno prav: ker smo na začetek dodali piko, so se vsi indeksi povečali za `1`.

Preverimo, ali to res deluje. Spomnimo se, vrstica je

[36]:
```
vrstica
```

[36]:
```
'##.##.####…#…#'
```

in pokličimo

[37]:
```
pretvori_vrstico(vrstica)
```

[37]:
```
[(1, 2), (4, 5), (7, 10), (14, 14), (19, 19)]
```

Zakaj delamo z indeksi, ne z `zip`, tako kot `prej`?

Lahko. Prvi razlog je, da vseeno potrebujemo indekse, torej nam `enumerate` ne uide. Iz tega sledi druga sitnost: enumerirali bomo zip. Zip bo naredil pare znakov, `enumerate` bo naredil pare indeksov in parov, ki jih je naredil `zip`. To bo zapletlo razpakiranje v `for`.

Vendar gre in za resnega programerja je to gotovo lepša rešitev.

[38]:
```
def pretvori_vrstico(vrstica):
    bloki = []
    for i, (prej, znak) in enumerate(zip("." + vrstica, vrstica + ".")):
        if prej + znak == ".#":
            zacetek = i + 1
        elif prej + znak == "#.":
            bloki.append((zacetek, i))
    return bloki
```

Zaresnemu programerju pa se s temi otročarijami ne da ukvarjati in uporabi regularne izraze (ki bi mu, mimogrede, tudi prejšnje naloge naredili trivialne).

[39]:
```
import re

def pretvori_vrstico(vrstica):
    return [(m.start() + 1, m.end()) for m in re.finditer("#+", vrstica)]
```

0.5 5. Pretvori zemljevid

`pretvori_zemljevid(vrstice)` prejme celoten zemljevid, kot ga vidimo v uvodnem delu naloge, in vrne seznam trojk `(x0, x1, y)`, ki predstavljajo ovire. Za gornji zemljevid mora vrniti `[(3, 4, 2), (2, 3, 3), (5, 5, 3), (4, 6, 4), (1, 3, 5), (5, 6, 5)]`. Seznam naj bo urejen po vrsticah in znotraj tega po stolpcih (tako kot v primeru). Pazi, tudi vrstice so oštevilčene od 1, ne 0.

0.5.1 Rešitev

Gremo čez oštevilčene vrstice. Vsako pretvorimo in v seznam ovir dodajamo dobljene koordinate začetkov in koncev, ki jim dodamo še vrstico.

[40]:
```
def pretvori_zemljevid(zemljevid):
    ovire = []
    for y, vrstica in enumerate(zemljevid, start=1):
        for x0, x1 in pretvori_vrstico(vrstica):
            ovire.append((x0, x1, y))
    return ovire
```

Tu se moramo predvsem upreti skušnjavi, da bi ponovno programirali to, kar smo že naredili v `pretvori_vrstico`. Potrebno je preprosto poklicati funkcijo, ki jo že imamo.

0.6 6. Izboljšave

`izboljsave(prej, potem)` prejme dva zemljevida - en je starejši, drugi novejši. Funkcija mora vrniti seznam novo postavljenih ovir; tudi ta mora biti urejen po vrsticah in znotraj tega po stolpcih. Mirno smeš predpostaviti, da MOL ne odstranjuje postavljenih ovir, temveč jih zgolj dodaja. Prav tako nobena ovira ni "razširjena"; nove ovire niso postavljene neposredno levo ali desno od obstoječih. Predpostaviti smeš tudi, da nova kolesarska steza ni širša. (No, tudi ožja ne; enake oblike je.)

0.6.1 Rešitev

Kako brez potrebe so si nekateri zapletli to nalogo! Primerjali vrstice iz obeh zemljevidov!

Čisto preprosto je: s `pretvori_zemljevid` dobimo sezname ovir prej in potem, ter ustvarimo nov seznam, ki vsebuje, kar je v `potem` in ni bilo v `prej`.

[41]:
```
def izboljsave(prej, potem):
    prej = pretvori_zemljevid(prej)
    potem = pretvori_zemljevid(potem)
```

```
    nove = []
    for ovira in potem:
        if ovira not in prej:
            nove.append(ovira)
    return nove
```

0.7 7. Huligani

`huligani(prej, potem)`; zgodi se, da pridejo huligani in kradejo ovire ter s tem ogrožajo varnost kolesarjev. Funkcija `huligani` naj zato vrne dva seznama: seznam vseh novih in seznam vseh odstranjenih ovir. Tu se lahko zgodi, da se nova ovira delno prekriva z neko prejšnjo.

0.7.1 Rešitev

Tule se je potrebno zgolj poglobiti v razmišljanje MOL in huliganov. Prometni strokovnjaki na Oddelku za gospodarstvo in motorni promet Mestne občine Ljubljana vedo, da je kolesarska steza izboljšana, če je na njej več ovir. Če je `potem` več ovir kot `prej`, je to torej `izboljsava`. Huligani zmotno mislijo, da je izboljšava, če ovira izgine, torej če je bilo `prej` več ovir kot `potem`.

Funkcija `huligani` mora vračati dve stvari. Prva je točno tisto, kar vrača funkcija `izboljsave`, drugo pa tisto, kar bi vrnila funkcija `izboljsave`, če bi - sledeč perverzni logiki sovražnikov ovir na kolesarskih stezah - obrnili čas nazaj.

[42]:
```
def huligani(prej, potem):
    return izboljsave(prej, potem), izboljsave(potem, prej)
```

0.8 Celotna rešitev

0.8.1 Elegantne, a razumne rešitve

Tule so zbrane rešitve, ki ne zahtevajo ničesar, česar se še nismo učili. V primerjavi z rešitvami iz gornjega besedila, smo poenostavili `najsirsa_ovira`, saj ni razloga, da ne bi klicala kar funkcije `pretvori_vrstica`. Tako se dejansko le enkrat zafrkavamo z zanko prek vrstice.

[43]:
```
def dolzina_ovir(vrstica):
    return vrstica.count("#")


def stevilo_ovir(vrstica):
    return ("." + vrstica).count(".#")


def najsirsa_ovira(vrstica):
    naj = 0
    for od, do in pretvori_vrstico(vrstica):
        if do - od + 1 > naj:
            naj = do - od + 1
    return naj
```

```
def pretvori_vrstico(vrstica):
    vrstica = "." + vrstica + "."
    bloki = []
    for i, znak in enumerate(vrstica):
        if znak == "#":
            if vrstica[i - 1] == ".":
                zacetek = i
            if vrstica[i + 1] == ".":
                bloki.append((zacetek, i))
    return bloki


def pretvori_zemljevid(zemljevid):
    ovire = []
    for y, vrstica in enumerate(zemljevid, start=1):
        for x0, x1 in pretvori_vrstico(vrstica):
            ovire.append((x0, x1, y))
    return ovire


def izboljsave(prej, potem):
    prej = pretvori_zemljevid(prej)
    potem = pretvori_zemljevid(potem)
    nove = []
    for ovira in potem:
        if ovira not in prej:
            nove.append(ovira)
    return nove


def huligani(prej, potem):
    return izboljsave(prej, potem), izboljsave(potem, prej)
```

0.9 Zaresne rešitve

Tole pa so lepe rešitve v pravem Pythonu. Tako bi z nalogo opravil profesionalec. Pokažem predvsem zato, da dobite malo občutka o tem, kako učinkovito je mogoče programirati v Pythonu.

[44]:
```
import re


def dolzina_ovir(vrstica):
    return vrstica.count("#")


def stevilo_ovir(vrstica):
```

```
    return ("." + vrstica).count(".#")


def najsirsa_ovira(vrstica):
    return max(od - do + 1 for od, do in pretvori_vrstico(vrstica))


def pretvori_vrstico(vrstica):
    return [(m.start() + 1, m.end()) for m in re.finditer("#+", vrstica)]


def pretvori_zemljevid(vrstice):
    return [(x0, x1, y)
            for y, vrstica in enumerate(zemljevid, start=1)
            for x0, x1 in pretvori_vrstico(vrstica)]


def izboljsave(prej, potem):
    prej = pretvori_zemljevid(prej)
    potem = set(pretvori_zemljevid(potem))
    return [ovira for ovira in prej if ovira not in potem]


def huligani(prej, potem):
    return izboljsave(prej, potem), izboljsave(potem, prej)
```

Morda opomba glede funkcije `izboljsave`: rešiti bi jo bilo možno v eni vrstici, kot razliko množic,

```
def izboljsave(prej, potem):
    return list(set(pretvori_zemljevid(potem)) - set(pretvori_zemljevid(prej)))
```

če množice ne bi pomešale vrstnega reda. Kasnejše urejanje v slogu

```
def izboljsave(prej, potem):
    return sorted(set(pretvori_zemljevid(potem)) - set(pretvori_zemljevid(prej)),
                  key=pretvori_zemljevid(potem).index)
```

pa je brez zveze. Najbrž ne pridobimo ničesar; metoda `index`, ki jo potrebujemo za to, da vrnemo elemente seznama v prvotni vrstni red, je počasna. Vse skupaj je le nepregledno. Je pa, seveda, zabavno.

Dodatek: eden od študentov je povedal, da deluje tudi

```
def izboljsave(prej, potem):
    return sorted(set(pretvori_zemljevid(potem)) - set(pretvori_zemljevid(prej)),
                  key=lambda x: x[2])
```

To uredi ovire po drugem elementu, vrstici. Mislim, da deluje slučajno: da bi zares delovalojih je treba urediti najprej po vrsticah, znotraj tega pa še po stolpcih,

```
def izboljsave(prej, potem):
```

```
    return sorted(set(pretvori_zemljevid(potem)) - set(pretvori_zemljevid(prej)),
                  key=lambda x: (x[2], x[0]))
```

To je v resnici hitrejše od vseh gornjih programov (ne da bi se pri tako majhnem številu ovir to kaj poznalo…). Deluje pa zato, ker vemo, da morajo biti ovire urejene po vrsticah in stolpcih. V splošnem pa: če imamo dva seznama in želimo dobiti seznam vsega, kar je v prvem, ne pa tudi v drugem, bomo z `list(set(s) - set(t))` izgubili vrstni red.

Rešitev, ki jo je našel študent, pa tu deluje in je seveda odlična ideja.