

Naloga se navezuje na nalogo Šikaniranje oziroma Lokacije ovir in bo zelo kratka. V vsaki funkciji bo potrebno napisati samo eno vrstico. Točneje, tokratna naloga bo krajša, ko bi si želeli: v vsaki funkciji boste *smeli* napisati samo eno vrstico. Vadimo namreč izpeljane sezname, množice in generatorje.

Funkcije se lahko kličejo med sabo, izjemoma pa ne smete pisati dodatnih funkcij, ki jih naloga ne zahteva.

Ovire bodo podane s seznamom trojk (y , x_0 , x_1), ki predstavljajo vrstico ter začetni in končni stolpec z oviro. Seznam ni urejen in je lahko videti, recimo, tako

```
[(1, 3, 6),  
 (1, 8, 10),  
 (2, 1, 4),  
 (3, 5, 8),  
 (2, 7, 9),  
 (7, 10, 10),  
 (7, 12, 13),  
 (5, 8, 10),  
 (5, 1, 3),  
 (2, 15, 19)]
```

Obvezna naloga

Napišite naslednje funkcije.

- `vrstice(ovire)` vrne seznam vseh vrstic z ovirami. Vrstice so naštet v enakem vrstnem redu kot v seznamu in se lahko ponavljajo. Za gornji primer funkcija vrne `[1, 1, 2, 3, 2, 7, 7, 5, 5, 2]`.
- `ovirane_vrstice(ovire)` vrača podoben seznam, le da je urejen. Za gornji primer vrne `[1, 1, 2, 2, 2, 3, 5, 5, 7, 7]`.
- `ovirane_vrstice_uni(ovire)` vrne podoben seznam, le da se vrstice ne ponavljajo. Za gornji primer vrne `[1, 2, 3, 5, 7]`. (Namig: v množico.)
- `ovire_v_vrstici(ovire, vrstica)` vrne množico začetkov in koncev vseh ovir v podani vrstici. Klic `ovire_v_vrstici(ovire, 2)` vrne `{(1, 4), (7, 9), (15, 19)}`.
- `stevilo_ovir(ovire, vrstica)` vrne število ovir v podani vrstici.
- `dolzina_ovir(ovire)` vrne skupno dolžino vseh ovir.
- `prosta_pot(ovire, stolpec)` vrne `True`, če nobena od ovir ne zapira podanega stolpca, in `False`, če takšna ovira obstaja.

Rešitev

`vrstice` mora vrniti seznam vseh indeksov y , ki se pojavijo v `ovire`.

```
def vrstice(ovire):  
    return [y for y, _, _ in ovire]
```

Naloga `ovirane_vrstice` na spomni, da obstaja funkcija `sorted`. Podamo ji lahko kar generator: vse, kar bo zgeneriral, bo zložila v seznam in ga uredila.

```
def ovirane_vrstice(ovire):  
    return sorted(y for y, _, _ in ovire)
```

Tretja naloga nas spomni, kako dobimo unikatne stvari: tako da jih vržemo v množico. Množico podamo `sorted`, ki bo elemente množice zložil v seznam in ga uredil.

```
def ovirane_vrstice_uni(ovire): # namig: v množico  
    return sorted({y for y, _, _ in ovire})
```

V četrti nalogi uporabimo pogoje: zanimajo nas začetki in konci, (x_0, y_0) v vseh tistih trojkah y, x_0, x_1 v seznamu `ovire`, za katere velja, da je y enak vrstica. Če to namesto v slovenščini povemo v Pythonu, dobimo:

```
def ovire_v_vrstici(ovire, vrstica):  
    return {(x0, x1) for y, x0, x1 in ovire if y == vrstica}
```

Število ovir v vrstici je lahko dolžina seznama, ki ga vrne funkcija `ovire_v_vrstici`.

```
def stevilo_ovir(ovire, vrstica):  
    return len(ovire_v_vrstici(ovire, vrstica))
```

Če te funkcije nimamo ali pa se želimo izogniti sestavljanju seznama, lahko uporabimo `sum`: $y == \text{vrstica}$ bo bodisi resničen, `True` ali neresničen, `False`, kar je enak 1 ali 0.

```
def stevilo_ovir(ovire, vrstica):  
    return len(ovire_v_vrstici(ovire, vrstica))
```

Dolžino ovir dobimo tako, da odštejemo $x_1 - x_0$ za vsako oviro - pa še 1 prištejemo, ker ovira vključuje tudi zgornjo mejo.

```
def dolzina_ovir(ovire):  
    return sum(x1 - x0 + 1 for _, x0, x1 in ovire)
```

Pot je prosta, če ni nobenega ovire, ki bi jo zapirala.

```
def prosta_pot(ovire, stolpec):  
    return not any(x0 <= stolpec <= x1 for _, x0, x1 in ovire)
```

De Morgan pa bi vedel povedati, da je to isto, kot če rečemo, da za vse ovire velja, da je ne zapirajo.

```
def prosta_pot(ovire, stolpec):  
    return all(not x0 <= stolpec <= x1 for _, x0, x1 in ovire)
```

Dodatne naloge

Tole so naloge za študentko iz približno pete vrste in za vse druge, ki jih zanima, zakaj je potrebno pri

```
[<izraz> for <spremenljivka> in <seznam> if <pogoj>]
```

razmišljati drugače kot pri

```
s = []
for <spremenljivka> in <seznam>:
    if <pogoj>:
        s.append(<izraz>)
```

Naloge niso težke, morda pa so poučne. Da bodo še poučnejše, predlagam, da izmed funkcij, ki ste jih sprogramirali zgoraj, v spodnjih funkcijah kličete kvečjemu funkcijo `ovirane_vrstice_uni`. Morda bo še boljše, če vse funkcije najprej sprogramirate s klasično zanko, da boste videli, zakaj takšne zanke ne morete preprosto predelati v izpeljan slovar. (Spoiler: ker ne morete "updateati" vrednosti v slovarju, ki ga še ni.)

- `stevila_ovir(ovire)` vrne slovar, katerega ključi so številke vrstic, ki vsebujejo vsaj eno oviro, vrednosti pa število ovir v tej vrstici.
- `zacetki(ovire)` vrne slovar, katerega ključi so številke vrstic z vsaj eno oviro, vrednosti pa najbolj levi ovirani stolpec.
- `ovire_po_vrsticah(ovire)` vrne slovar, katerega ključi so številke vrstic z vsaj eno oviro, vrednosti pa množice začetkov in koncev ovir v tej vrstici.

Rešitev

Številna ovir Brez izpeljanega slovarja bi prvo nalogo rešili tako:

```
from collections import defaultdict

def stevila_ovir(ovire):
    stevila = defaultdict(int)
    for y, _, _ in ovire:
        stevila[y] += 1
    return stevila
```

Zato pridemo na idejo, da bi se na podoben način lotili tudi z izpeljanim slovarjem.

```
from collections import defaultdict

def stevila_ovir(ovire):
    return {y: sum(1 for _, _ in ovire[y]) for y in ovire}
```

Problem je, kajpa, v `sum(1 for _, _ in ovire[y])`. K čemu bomo prišteli 1? Tega slovarja še ni.

Zato sem na predavanjih poudarjal, da ta slovar nastane takšen, kot je. Med nastajanjem ga ne moremo spreminjati, se sklicevati nanj... Rešitev v eni vrstici bo morala biti

```
from collections import defaultdict
```

```
def stevila_ovir(ovire):b
    return {y: !?!?! for y, _, _ in ovire}
```

pri čemer bo !?!?! že dokončna vrednost, število vseh ovir v podani vrstici.

Napisati moramo torej

```
def stevila_ovir(ovire):
    return {y: stevilo_ovir(ovire, y)
            for y in ovire}
```

Če funkcije `stevilo_ovir` še ne bi imeli, pa bi število ovir pač izračunali prav tu:

```
def stevila_ovir(ovire):
    return {y: sum(yo == y for yo, _, _ in ovire)
            for y in ovire}
```

Tu je nerodno, da se to računa za vsako oviro, torej bomo za vsako vrstico tolikokrat šteli ovire, koliko ovir je v tej vrstici. To ne bo vplivalo na končni rezultat, bo pa, seveda, na hitrost. Temu se izognemo tako, da gremo čez množico oviranih vrstic.

```
def stevila_ovir(ovire):
    return {y: stevilo_ovir(ovire, y)
            for y in ovirane_vrstice(ovire)}
```

Ali, če nobene od teh dveh funkcij še ne bi imeli:

```
def stevila_ovir(ovire):
    return {y: sum(yo == y for yo, _, _ in ovire)
            for y in {y for y, _, _ in ovire}}
```

Treba pa je pošteno povedati, da je ta rešitev verjetno počasnejša od prve, tiste z običajno zanko, saj se zanka `for yo, _, _ in ovire` tu izvede tolikokrat, kolikor je zasedenih vrstic.

Začetki Tole je ista finta, le da namesto funkcije `stevilo_ovir` iščemo minimume.

```
def zacetki(ovire):
    return {y: min(x0 for yo, x0, _ in ovire if y == yo)
            for y in ovirane_vrstice(ovire)}
```

Ovire po vrsticah In tole je spet isto, le da tokrat sestavljamo sezname.

```
def ovire_po_vrsticah(ovire):  
    return {  
        y: {(x0, x1) for yo, x0, x1 in ovire if yo == y}  
        for y in ovirane_vrstice(ovire)}
```

Če bi najprej sprogramirali `ovire_po_vrsticah`, ostali dve funkciji izpeljemo iz nje:

```
def stevila_ovir(ovire):  
    return {y: len(x0x1)  
            for y, x0x1 in ovire_po_vrsticah(ovire).items()}  
  
def zacetki(ovire):  
    return {y: min(x for x, _ in x0x1)  
            for y, x0x1 in ovire_po_vrsticah(ovire).items()}
```