

Ovire so podane kot seznam terk (x_0 , x_1 , y), kjer je y številka vrstice, x_0 in x_1 pa skrajni levi in desni stolpec ovire. Seznam ovir ni urejen ne po vrsticah ne kakorkoli drugače.

Ovire na sliki (ki se uporabljajo v nalogi 3, z nekaj dodatki pa še v 1 in 2) so lahko predstavljene s seznamom so lahko predstavljene s seznamom [(1, 3, 6), (2, 4, 3), (3, 4, 9), (6, 9, 5), (9, 10, 2), (9, 10, 8), (4, 6, 7)].

1. Nestrinjanja

Dva vestna uslužbenca MOL sta pripravila dva predloga seznamov ovir. Nekatere ovire sta predlagala oba, nekatere pa le eden ali drugi. Da bodo lažje uskladili predlog in čimprej postavil nujno potrebne ovire za kolesarje, napiši funkcijo `nestrinjanja(ovire1, ovire2)`, ki prejme dva seznama ovir in vrne množico vseh ovir, ki se pojavijo v enem ali v drugem seznamu, ne pa v obeh.

Rešitev

Tole je naloga iz množic in rešitev je

```
def nestrinjanja(ovire1, ovire2):  
    return set(ovire1) ^ set(ovire2)
```

Kdor se ne spomni operatorja `^`, ki računa simetrično razliko (torej, kar je v eni množici in ne v obeh), je lahko naredil unijo (kar je predlagal kdorkoli, ne pa oba) in odštél presek (kar sta predlagala oba).

```
def nestrinjanja(ovire1, ovire2):  
    return (set(ovire1) | set(ovire2)) - (set(ovire1) & set(ovire2))
```

Kdor pa se sploh ni spomnil na množice in operacije z njimi, je moral napisati kaj takšnega:

```
def nestrinjanja(ovire1, ovire2):  
    razlika = set()  
    for x in ovire1:  
        if x not in ovire2:  
            razlika.add(x)  
    for x in ovire2:  
        if x not in ovire1:  
            razlika.add(x)  
    return razlika
```

Sam si je kriv.

2. Proste ovire

Napiši funkcijo `proste_ovire(ovire)`, ki dobi seznam ovir, med katerimi pa se nekatere prekrivajo (ali pa celo popolnoma sovpadajo). Funkcija naj vrne

nov seznam, v katerem so le ovire, ki se ne prekrivajo (delno ali popolnoma) z nobeno drugo oviro.

Rešitev

Prva naloga vam je bila podarjena, ker je bila res preprosta. Ta naloga je pa malo bolj (ampak ne zelo) zoprna. Kot sem napovedal, je na vsakem izpitu še kakšna naloga, kjer morate znati pametno preobrniti kakšno zanko.

Napisali bomo ločeno funkcijo `prekrivanje(ovira1, ovira2)`, ki bo povedala, ali se dve oviri prekrivata. To sploh ni nujno; namenjeno je le temu, da bo "glavna" funkcija preglednejša.

```
def prekrivanje(ovira1, ovira2):
    x10, x11, y1 = ovira1
    x20, x21, y2 = ovira2
    return y1 == y2 and (x20 <= x10 <= x21 or x10 <= x20 <= x11)
```

Malo preprostejša, vendar počasnejša je rešitev, ki za vsako oviro preveri vse ovire; če naleti na oviro, ki ni prav taista ovira (`i != j`) in se prekriva, prekine zanko (`break`). Če se zanka izteče, ne da bi jo prekinili (`else`), dodamo oviro v seznam prostih.

```
def proste_ovire(ovire):
    proste = []
    for i, ovira1 in enumerate(ovire):
        for j, ovira2 in enumerate(ovire):
            if i != j and prekrivanje(ovira1, ovira2):
                break
        else:
            proste.append(ovira1)
    return proste
```

Rešitev je nekoliko počasnejša zato, ker gre za vsako oviro čez vse druge ovire. Lahko bi za vsako oviro preverili vse ovire pred njo; če se prekrivata, bi obe označili kot ne-prosti. Takšna funkcija bo torej najprej pripravila seznam "zastavic", ki bo dolg toliko, kolikor je ovir. Za vsako oviro bo vseboval `True`, če je prosta in `False`, če ni. V začetku bo seznam vseboval same `True`, potem pa bomo šli čez vse ovire in za vsako preverili, ali jo prekriva katera od ovir, ki ji sledijo. Če jo, obema spremenimo njun element v `False`.

Na koncu zipnemo skupaj seznam ovir in seznam zastavic. V nov seznam damo oviro, katere pripadajoča zastavica je `True`.

```
def proste_ovire(ovire):
    proste = [True] * len(ovire)
    for i, ovira1 in enumerate(ovire):
        for j in range(i):
            if prekrivanje(ovira1, ovire[j]):
```

```

        proste[i] = proste[j] = False
    return [ovira for ovira, prosta in zip(ovire, proste) if prosta]

```

Ta naloga je bila nekoliko težja, vendar lažja od prvotne različice, ki je zahtevala, da funkcija briše ovire iz obstoječega seznama. To vam je bilo torej prihranjeno ... lahko pa poskusite to narediti za vajo.

3. Dolžina ovir

Napiši funkcijo `dolzina_ovir(ime_datoteke)`, ki prejme ime datoteke, v kateri so ovire opisane v takšni obliki:

```

1-3 6
2-4 3
3-4 9

```

(in tako naprej). Prvi števili sta začetek in konec ovire, ločena z `-`. Sledi presledek in številka vrstica. Funkcija naj vrne skupno dolžino ovir (število "pobarvanih" kvadratkov na sliki). Ovire se ne prekrivajo.

Rešitev

Za oddih spet nekoliko lažja naloga. Znati moramo odpreti datoteko, jo brati po vrsticah, razdeliti niz glede na presledek, razdeliti del tega niza glede na `-`, spremeniti niza v števili, ter ju odšteti in prištrevati k skupni vsoti. Eh.

```

def dolzina_ovir(ime_datoteke):
    dolzina = 0
    for vrstica in open(ime_datoteke):
        xs, _ = vrstica.split()
        x0, x1 = xs.split("-")
        dolzina += int(x1) - int(x0) + 1
    return dolzina

```

Številke vrstice ne potrebujemo. :) Upam, da reševalcem ni vzelo preveč časa, da so odkrili, da morajo k razliki med stolpcema prišteti 1. Če jim je, je to zgolj poučno: bodo bolj cenili, da je indeksiranje v Pythonu narejeno tako, da `s[5:8]` vsebuje 3 elemente (8 - 5) in ne štirih (od petega do *inključno* osmega), tako kot ovire.

4. Združevanje ovir

Seznam, kot je, na primer, [(1, 3, 2), (4, 6, 2), (4, 8, 5), (11, 13, 5), (1, 3, 6), (3, 5, 8), (6, 6, 8), (7, 10, 8)], je možno poenostaviti tako, da združimo ovire, ki se stikajo. Tako je na primer (1, 3, 2), (4, 6, 2) možno zamenjati z (1, 6, 2). Prav tako je (3, 5, 8), (6, 6, 8), (7, 10, 8) možno zamenjati z (3, 10, 8).

Napiši **rekurzivno funkcijo** `zdruzi_ovire(ovire)`, ki prejme seznam v tej obliki in vrne seznam, v katerem so ovire, ki se stikajo, združene. V gornjem primeru vrne `[(1, 6, 2), (4, 8, 5), (11, 13, 5), (1, 6, 6), (7, 10, 8)]`.

Oviri se stikata, če sta v isti vrstici in se ena začne takoj po koncu druge. Prekrivanj ni.

Predpostaviti smete, da bo podani seznam **urejen** po vrsticah in, znotraj vrstic, po stolpcih. **Ne smete pa predpostaviti**, da se vedno združita le po dve ali tri ali štiri ... ali kako vnaprej predpisano omejeno število ovir.

Enigmatični namig: funkcija naj k ostanku ovir doda ali pridruži prvo.

Rešitev

Rekurzija kot rekurzija. Malo drugače moraš pomisliti, pa je preprosta. Če ne pomisliš drugače, pa je nerešljiva.

Pri tej nalogi je zanimivo, da smo se s kolegi pogovarjali, da jo je pravzaprav lažje, lepše rešiti rekurzivno.

Če imamo eno oviro (ali nič), ni česa združevati: seznam vrnemo tak, kot je.

Sicer z rekurzivnim klicem združimo vse ovire, razen prve. Nato primerjamo prvo oviro seznama ovir (to, ki smo jo prej preskočili) in prvo oviro ostanka. Če se stikata, vrnemo seznam, v katerem staknemo tidve oviri `[(x00, x11, y0)]` in ostale ovire. Če se ne stikata, pa vrnemo seznam, ki vsebuje prvo oviro in tudi vse ostale.

```
def zdruzi_ovire(ovire):
    if len(ovire) <= 1:
        return ovire

    ostanek = zdruzi_ovire(ovire[1:])

    x00, x01, y0 = ovire[0]
    x10, x11, y1 = ostanek[0]
    if y0 == y1 and x10 == x01 + 1:
        return [(x00, x11, y0)] + ostanek[1:]
    return [ovire[0]] + ostanek
```

5. Kolesarska

Napiši razreda `Kolesarska` in `InteligentnaKolesarska`. Razred `Kolesarska` ima, poleg konstruktorja, metode:

- `dodaj_oviro(x0, x1, y)`, ki doda oviro na podano mesto,
- `stevilo_ovir()` vrne število vseh ovir
- `prosto(x, y)` vrne `True`, če je podano polje prosto.

Razred `InteligentnaKolesarska` naj bo izpeljan iz razreda `Kolesarska`. Spreminja le metodo `prosto` in sicer tako, da ob vsakem klicu sicer vrne `True` ali `False`, poleg tega pa na to polje postavi oviro (velikosti 1, zaseda le to polje), tako polje ob naslednjem klicu ne bo več prosto.

Bistvo naloge je, da mora metoda `prosto` primerno uporabljati podedovane metode. To bo tudi točkovano.

Rešitev

Prvi razred je preprost. Konstruktor pripravi prazen seznam ovir. `dodaj_oviro` vanj dodaja ovire, `stevilo_ovir` vrne dolžino seznama, `prosto` pa preveri, ali obstaja kaka ovira, ki pokriva podano polje.

```
class Kolesarska:
    def __init__(self):
        self.ovire = []

    def dodaj_oviro(self, x0, x1, y):
        self.ovire.append((x0, x1, y))

    def stevilo_ovir(self):
        return len(self.ovire)

    def prosto(self, x, y):
        return not any(yo == y and x0 <= x <= x1 for x0, x1, yo in self.ovire)
```

Da sprogramiramo drugega, moramo znati povoziti podedovano metodo, pa še malenkost spretni moramo biti: novo oviro dodamo šele po tem, ko s klicem podedovane metode preverimo, ali je podano polje prosto.

```
class InteligentnaKolesarska(Kolesarska):
    def prosto(self, x, y):
        je_prosto = super().prosto(x, y)
        if je_prosto:
            self.dodaj_oviro(x, x, y)
        return je_prosto
```

Z "walrusom", ki ga pri Programiranju 1 nisem kazal, lahko naredimo tudi tako:

```
class InteligentnaKolesarska(Kolesarska):
    def prosto(self, x, y):
        if (je_prosto := super().prosto(x, y)):
            self.dodaj_oviro(x, x, y)
        return je_prosto
```

Eden od študentov je naredil napisal takole:

```
class InteligentnaKolesarska(Kolesarska):
    def prosto(self, x, y):
```

```

    if super().prosto(x, y):
        self.dodaj_oviro(x, x, y)
        return True
    return False

```

To je pravzaprav tudi čisto elegantno.

Tipičen programer v javascriptu pa bi napisal

```

class InteligentnaKolesarska(Kolesarska):
    def prosto(self, x, y):
        return super().prosto(x, y) and (self.dodaj_oviro(x, x, y) or True)

```

Mi, Pythonaši, pa smo lepo vzgojeni in raje napišemo kakšno vrstico več, namesto da bi tem, ki berejo naše programe, zastavljali takšne nepotrebne uganke. (Včeraj sem namreč delal nekaj v Reactu in napisal tale košček javascripta: `{ !!children && children }`. V Pythonu mi to ne bi prišlo na misel.