

Kot v prvih domačih nalogah bodo ovire predstavljene s trojkami (x_0, x_1, y) , pri čemer se ovira razprostira od stolpca x_0 do vključno x_1 , šteti pa začnemo z 1. Zemljevid na sliki bi opisali z $[(4, 7, 1), (1, 3, 2), (10, 14, 2), (4, 7, 4), (10, 14, 4), (4, 7, 6), (1, 3, 7), (6, 7, 7), (11, 15, 7)]$.

```

1
123456789012345
1 ...<-->.....
2 <->.....<--->.
3 .....
4 ...<-->..<--->.
5 .....
6 ...<-->.....
7 <->..<>...<--->

```

1. Nabava

Zemljevidi vsebujejo ovire različnih dolžin. Gornji zemljevid, recimo, ima 1 oviro dolžine 2, 2 oviri dolžine 3, 3 ovire dolžine 4 in 3 ovire dolžine 5. Nekega dne se MOL odloči spremeniti razpored ovir na neki kolesarski poti, zato bo morda potrebno nabaviti nekaj novih ovir določenih dolžin (kar bo sicer strošek, vendar MOL takrat, ko gre za varnost kolesarjev, ne varčuje!)

Napiši funkcijo `nabava(stari, novi)`, ki prejme dva seznama ovir in vrne slovar, katerega ključi so dolžine ovir, ki jih bo potrebno dokupiti, ker med obstoječimi ovirami ni (dovolj) ovir takšne dolžine. Pripadajoče vrednosti bodo število potrebnih novih ovir. Če je potrebno dokupiti 3 ovire dolžine 5 in sedem ovir dolžine 1, funkcija vrne `{5: 3, 1: 7}`.

Rešitev

To je naloga iz slovarjev. Če se spomnimo nanje, je lahko čisto preprosta.

Ena rešitev je lahko takšna, da v slovar preštejemo vse, kar potrebujemo (seveda bomo uporabili `defaultdict`. Nato od njega odštevamo, kar imamo. Na koncu vrnemo slovar, ki vsebuje le elemente s pozitivno vrednostjo.

```

from collections import defaultdict

def nabava(staro, novo):
    ovire = defaultdict(int)
    for x0, x1, _ in novo:
        ovire[x1 - x0 + 1] += 1
    for x0, x1, _ in staro:
        ovire[x1 - x0 + 1] -= 1
    return {k: v for k, v in ovire.items() if v > 0}

```

Kozmetični detajl: čez slovar gremo z `for x0, x1, _ in`. Koordinate razpakiramo že v slovarju in nepotrebno številko vrstice damo v spremenljivko z imenom `_`.

Sitnost je v zadnji vrstici. Morda bi koga zamikalo napisati

```
for dolzina, stevilo in ovire.items():
    if stevilo <= 0:
        del ovire[dolzina]
return ovire
```

Vendar bi ga čakalo razočaranje: če gremo z zanko čez slovar, znotraj te zanke ne moremo spreminjati (dolžine) tega slovarja. Tu moramo narediti nov slovar, kot smo ga zgoraj ali pa, po daljši poti z

```
kupiti = {}
for dolzina, stevilo in ovire.items():
    if stevilo > 0:
        kupiti[dolzina] = stevilo
return kupiti
```

Druga možnost je, da zanke ne poženemo pred slovarja temveč prek seznama, ki vsebuje vse, kar je v tem slovarju:

```
for dolzina, stevilo in list(ovire.items()):
    if stevilo <= 0:
        del ovire[dolzina]
return ovire
```

Vseeno - najboljše je prvo.

Daljša rešitev je, da ločeno preštejemo, kaj potrebujemo in kaj imamo, potem pa sestavimo nov slovar, v katerega dajemo razliko med obema.

```
def nabava(staro, novo):
    potrebe = defaultdict(int)
    for x0, x1, _ in novo:
        potrebe[x1 - x0 + 1] += 1
    imamo = defaultdict(int)
    for x0, x1, _ in staro:
        imamo[x1 - x0 + 1] += 1
    kupiti = {}
    for dolzina, stevilo in potrebe.items():
        if stevilo > imamo.get(dolzina, 0):
            kupiti[dolzina] = stevilo - imamo.get(dolzina, 0)
    return kupiti
```

2. Rekonstrukcija

Napiši funkcijo `rekonstrukcija(kocke)`, ki dobi seznam polj, ki jih pokrivajo ovire. Seznam je podan v obliki parov (y, x) (pazi: najprej y , potem x , saj to olajša nalogo!), ki pa niso nujno urejeni. Funkcija mora vrniti seznam dejanskih ovir v obliki običajnih trojk. Vrnjeni seznam mora biti urejen po vrsticah in znotraj vrstic po stolpcih.

Klic `rekonstrukcija([(2, 3), (1, 1), (2, 2), (2, 4), (1, 2), (3, 4)])` vrne `[(1, 2, 1), (2, 4, 2), (4, 4, 3)]`.

Rešitev

Vsak izpit ima (ali vsaj naj bi imel) nalogo, pri kateri je potrebno malo vozlati zanke in pogoje. Na tem izpitu je to ta.

```
def rekonstrukcija(kocke):
    ovire = []
    prvi_x = zadnji_x = zadnji_y = None
    for y, x in sorted(kocke):
        if prvi_x is None:
            prvi_x = x
        elif y != zadnji_y or x != zadnji_x + 1:
            ovire.append((prvi_x, zadnji_x, zadnji_y))
            prvi_x = x
        zadnji_x, zadnji_y = x, y
    if prvi_x is not None:
        ovire.append((prvi_x, zadnji_x, zadnji_y))
    return ovire
```

`ovire` je očitno seznam, v katerega zložimo ovire in ga na koncu vrnemo.

Seznam kock je potrebno urediti. Brez tega ... bo težka. Imeli bomo torej zanko `for y, x in sorted(kocke)`. V tej zanki moramo za vsako novo kocko vedeti, ali je del obstoječe ovire, ali pa je ovire, ki smo jo pregledovali v zadnjih korakih zanke, zdaj konec in jo je potrebno shraniti. Potrebujemo torej naslednje spremenljivke

- `prvi_x` je začetek tekoče ovire. V začetku je `None` in vsakič, ko naletimo na novo oviro, ga postavimo na začetek le-te.
- `zadnji_x` in `zadnji_y` sta `x` in `y` iz prejšnjega koraka zanke. Trenutna kocka nadaljuje prejšnjo oviro, če je trenutni `x` za 1 večji od zadnjega, trenutni `y` pa je enak zadnjemu.

V zanki se dogaja to: če je `prvi_x` enak `None`, gre za prvi korak zanke. Ta kocka je začetek prve ovire.

Sicer preverimo, ali se s trenutno kocko začenja nova ovira. Kdaj je to, smo opisali zgoraj. V tem primeru shranimo v seznam oviro, ki se je s prejšnjim

korakom zanke končala in si zapomnimo trenutni `x` kot začetek nove ovire.

Na koncu shranimo trenutna `x` in `y` kot `zadnji_x` in `zadnji_y`, da nam bosta dostopna v naslednjem koraku.

Po zanki je še zoprna zafrkancija: ker se prejšnja ovira ni končala z novo kocko, jo je potrebno dodati posebej. Seveda, če obstaja - če je `prvi_x` še vedno `None`, na stezi ni nobenih ovir.

Seveda obstaja še veliko drugih rešitev, ena je recimo ta:

```
def rekonstrukcija(kocke):
    if not kocke:
        return []
    ovire = []
    prvi_x = None
    kocke = sorted(kocke)
    for (y, x), (nasl_y, nasl_x) in zip(kocke, kocke[1:] + [(0, 0)]):
        if prvi_x is None:
            prvi_x = x
        if nasl_y != y or nasl_x != x + 1:
            ovire.append((prvi_x, x, y))
            prvi_x = nasl_x
    return ovire
```

Zanimiva - in kar kratka in razumljiva - je tale: vse kocke spremenimo v ovire. Potem gremo po seznamu teh "ovir" in ovire, ki se stikajo, združimo.

```
def rekonstrukcija(kocke):
    ovire = [(x, x, y) for y, x in sorted(kocke)]
    i = 1
    while i < len(ovire):
        if ovire[i - 1][2] == ovire[i][2] and ovire[i - 1][1] + 1 == ovire[i][0]:
            ovire[i - 1] = (ovire[i - 1][0], ovire[i][1], ovire[i][2])
            del ovire[i]
        else:
            i += 1
    return ovire
```

Pogoje preverja, ali se `i`-ta ovira stika s prejšnjo. Če se, ju združi in pobriše `i`-to. Sicer pa poveča `i`.

Pa še ena: mislim, da bi jo imeli matematiki radi zaradi

```
vrstica = [x - i for i, x in enumerate(sorted(vrstica))]
dolzine = Counter(vrstica)
```

Nadebudnji študenti lahko razmislijo, zakaj in kako to deluje.

```
def rekonstrukcija(kocke):
    ovire = []
```

```

vrstice = defaultdict(list)
for y, x in kocke:
    vrstice[y].append(x)
for y, vrstica in sorted(vrstice.items()):
    vrstica = [x - i for i, x in enumerate(sorted(vrstica))]
    dolzine = Counter(vrstica)
    n = 0
    for x, dolzina in sorted(dolzine.items()):
        ovire.append((x + n, x + n + dolzina - 1, y))
        n += dolzina
return ovire

```

3. Spet nov format

Napiši funkcijo `dekodiraj_vrstico(vrstica)`, ki dobi eno vrstico zemljevida v obliki, kot ga vidiš na vrhu izpita. Vrne seznam začetkov in koncev ovir: `dekodiraj_vrstico("..<-->...<---->..<>")` vrne `[(3, 6), (10, 15), (18, 19)]`. Ovir dolžine 1 ni. Je Angelca rekla, da so čisto brez zveze.

Napiši funkcijo `preberi(ime_datoteke)`, ki prejme ime datoteke, v kateri je shranjen zemljevid v takšni obliki. Vrne naj seznam ovir (urejen po vrsticah, znotraj po stolpcih). Za zemljevid z gornje slike mora vrniti seznam iz besedila ob njej.

Rešitev

Tretja naloga preverja, kako smo kaj domači z nizi in datotekami. In ne bi smela biti pretežka.

Najočitnejša rešitev prvega dela je

```

def dekodiraj_vrstico(vrstica):
    ovire = []
    for x, c in enumerate(vrstica, start=1):
        if c == "<":
            prvi = x
        elif c == ">":
            ovire.append((prvi, x))
    return ovire

```

Malo bolj zabavno je pobrati indekse vseh < in vseh > ter jih zipniti v en seznam.

```

def dekodiraj_vrstico(vrstica):
    return list(zip((i for i, c in enumerate(vrstica, start=1) if c == "<"),
                    (i for i, c in enumerate(vrstica, start=1) if c == ">")))

```

Da preberemo datoteko, zgolj pokličemo to funkcijo za vse vrstice ter v seznam ovir dodajamo trojke, ki vsebujeta začetek in konec ovire ter številko vrstice.

```
def preberi(ime_datoteke):
    ovire = []
    f = open(ime_datoteke)
    for y, vrstica in enumerate(f, start=1):
        ovire += ((x0, x1, y) for x0, x1 in dekodiraj_vrstico(vrstica))
    return ovire
```

4. Sodobna umetnost

V sklopu konference Velocity je potekala tudi umetniška razstava z inštalacijo iz ovir. Naložili so jih, kot kaže slika na desni: na zelo široki oviri d stojita oviri a in b. Na a stojita c in r ... in tako naprej. Takšno postavitve predstavimo s slovarjem `{" ": "dh", "d": "ab", "h": "tef", "a": "cr", "b": "uv", "t": "xy", "f": "qm", "c": "w", "r": "i", "u": "o", "v": "p", "x": "s", "q": "g", "m": "n", "w": "j", "o": "l", "s": "z", "g": "B", "n": "A", "l": "T"}`. Ovira je vrh, če na njej ni nobene druge ovire. V primeru na sliki so vrhovi j, i, T, p, z, y, e, B, A.

Napiši funkcijo `vrhovi(skladovnica, ovira, visina)`, ki prejme takšen slovar, neko oviro in neko višino. Vrniti mora množico imen vrhov, ki so nad to oviro in so za vsaj visina višji od nje. Funkcija naj bo seveda splošna in naj ne deluje le za ovire v tej umetnini. Seveda pa predpostavimo, da je ime ovire vedno podano z eno samo črko. Klic `vrhovi(skladovnica, "a", 3)` vrne `{"j"}`, in `vrhovi(skladovnica, "a", 2)` vrne `{"j", "i"}`.

Namig: višina je lahko tudi negativna. V tem primeru funkcija vrne vse vrhove nad oviro. (To ti bo v pomoč!)

Rešitev

To je seveda naloga iz naše ljube rekurzije.

```
def vrhovi(skladovnica, ovira, visina):
    ovire = set()
    if ovira not in skladovnica:
        if visina <= 0:
            ovire.add(ovira)
    else:
        for nadovira in skladovnica[ovira]:
            ovire |= vrhovi(skladovnica, nadovira, visina - 1)
    return ovire
```

Pripravimo si množico, ki jo bomo vrnili. Če ovire ni v slovarju, ki opisuje, kaj je na čem, jo dodamo, vendar le, če je dovolj visoko (`visina <= 0`). Sicer pa v množico ovir dodamo vse ovire, ki stojijo na tej oviri in so visoke vsaj `visina - 1`.

5. Potapljanje ovir

Napiši razred Ovire.

- Konstruktor sprejme množico ovir, podanih s trojkami (x0, x1, y).
- Metoda strel(x, y) prejme koordinate nekega polja. Če je tam ovira, vrne True, sicer False.
- Metoda zadetkov() vrne število strel, ki so zadeli kako oviro.
- Metoda vse_ovire() vrne ovire, ki še obstajajo. Ovira, ki je trikrat zadeta (lahko trikrat v isto točko, lahko v različne), se razblini (v celoti, ne le tam, kjer je bila zadeta).
- Metoda zmaga() vrne True, če ni ostala nobena ovira več. Sicer pa False.

Rešitev

Pri objektnem programiranju je - vsaj pri osnovnih nalogah, ki jih rešujemo pri Programiranju 1 - "najtežji" korak, da se odločimo, kako bomo shranjevali podatke. Tule očitno potrebujemo število zadetkov in nekaj, kamor bomo shranili vse ovir, ki še obstajajo. Ker moramo za vsako oviro vedeti tudi, kolikokrat je bila zadeta, bomo ovire shranjevali v slovar - ovire bodo ključi, pripadajoče vrednosti pa število zadetkov.

```
class Ovire:
    def __init__(self, ovire):
        self.zadetkov_ = 0
        self.ovire = dict.fromkeys(ovire, 0)

    def strel(self, x, y):
        for ovira in self.ovire:
            if ovira[0] <= x <= ovira[1] and ovira[2] == y:
                self.zadetkov_ += 1
                self.ovire[ovira] += 1
                if self.ovire[ovira] == 3:
                    del self.ovire[ovira]
            return True
        return False

    def zadetkov(self):
        return self.zadetkov_

    def vse_ovire(self):
        return set(self.ovire)

    def zmaga(self):
        return not self.ovire
```

Slovar smo naredili z `dict.fromkeys(ovire, 0)`. Kaj naredi, je najbrž očitno.

Če tega ne poznamo, pač pišemo

```
self.ovire = {}  
for ovira in ovire:  
    self.ovire[ovira] = 0
```

Edina metoda, s katero imamo v resnici kaj dela, je **strel**. Ta gre čez slovar ovir in če ugotovi, da smo kaj zadeli, poveča število zadetkov v splošnem ter število zadetkov te ovire. Če je ovira s tem zadeta tretjič, jo odstranimo in vrnemo **True**.

Tule bi lahko kdo rekel - kaj pa je s tistim, da znotraj zanke čez slovar ne smemo spreminjati tega slovarja? To še vedno velja, vendar smo tule zanko prekinili, torej ne bo nič narobe.

Ostale metode zgolj vrnejo, kar morajo.

Omenimo le še, da smo atribut s številom zadetkov poimenovali **zadetkov_**, da se njegovo ime ne tepe z imenom metode **zadetkov**.