

Mestna občina Ljubljana je objavila video o vedenju kolesarjev v Ljubljani. Naj povzamem: kolesarji so znani po divjih spustih po stopnicah, divjanju med pešci in tako naprej.* Ker je osnovno prevozno sredstvo vašega profesorja kolo in ker tretjino letne kilometrine žal opravi v Ljubljani, vas prosi, da mu za lažje načrtovanje poti rešite tole nalogo.

* Seveda se med kolesarji v resnici najdejo tudi breobzirni divjaki. Vtis, ki ga želi pustiti video, pa je vseeno morda nekoliko pretiran. Sploh začetno divjanje po stopnicah; downhill vidimo na Golovcu, Klobuku in Rašici, ne ob Ljubljani. Pa tudi tam večina kolesarjev spodobno pazi na pešce, slab vtis puščajo le posamični norci.

Oddelek za gozdarske dejavnosti in motorni promet MOL je pripravil zemljevid na sliki. Ta kaže 21 križišč v Ljubljani, pri čemer so na zahtevo mestnih strokovnjakov za varstvo osebnih podatkov zamenjali imena lokacij s črkami od A do V. Povezave med njimi zahtevajo različne veščine: kdor hoče, na primer priti iz točke B do C, mora obvladati vožnjo med odvrženimi skiroji in slalom med cvetličnimi lonci.

Celoten seznam veščin, ki se pojavljajo v nalogi, je:

- stopnice: Spust po stopnicah
- pešci: Divjanje med pešci
- lonci: Slalom med cvetličnimi lonci
- bolt: Slalom med odvrženimi skiroji
- robnik: Skok na robnik pločnika
- gravel: Vožnja po razsutem makadamu
- trava: Oranje zelenic parkov
- avtocesta: Vožnja po avtocesti
- crepinje: Vožnja po razbiti steklovini
- rodeo: Vožnja po kolesarski poti skozi Črnuče

Zemljevid na sliki zaradi pomanjkanja prostora uporablja enočrkovne okrajšave veščin, v sami nalogi pa je zapisan takole:

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, R, S, T, U, V = \ "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

```
zemljevid = {
    ('A', 'B'): {'trava', 'gravel'},
    ('A', 'V'): {'lonci', 'pešci'},
    ('B', 'A'): {'trava', 'gravel'},
    ('B', 'C'): {'lonci', 'bolt'},
    ('B', 'V'): set(),
    ('C', 'B'): {'lonci', 'bolt'},
    ('C', 'R'): {'lonci', 'pešci', 'stopnice'},
```

```

('D', 'F'): {'pešci', 'stopnice'},
('D', 'R'): {'pešci'},
('E', 'I'): {'trava', 'lonci'},
('F', 'D'): {'pešci', 'stopnice'},
('F', 'G'): {'trava', 'črepinje'},
('G', 'F'): {'trava', 'črepinje'},
('G', 'H'): {'črepinje', 'pešci'},
('G', 'I'): {'avtocesta'},
('H', 'G'): {'črepinje', 'pešci'},

```

... in tako naprej

Ključni zemljevidi so pari povezanih točk, pripadajoča vrednost pa je množica veščin, ki jih mora obvladati kolesar, če želi prevoziti to povezavo. Tako vidimo pod ključem (B, C) zapisano {"bolt", "lonci"}, kar je okrajšava za veščini *Slalom med odvrženimi skiroji* in *Slalom med cvetličnimi lonci*.

Vse povezave so dvosmerne, saj je kolesarjem po mnenju MOL itak vseeno, v katero smer in po kateri strani ceste vozijo. Če obstaja v slovarju povezava (B, C), torej obstaja tudi (C, B); obe zahtevata enake veščine. (Podvojene povezave so v bistvu nepotrebne, vendar vam bo tako lažje programirati.)

Obvezna naloga

Napiši naslednje funkcije

- `mozna_pot(pot, zemljevid)` prejme `pot` v obliki niza z zaporedjem križišč in `zemljevid` v obliki iz uvoda naloge. Funkcija mora vrniti `True`, če je takšna pot možna (torej: če obstajajo povezave med vsemi zaporednimi križišči v nizu) in `False`, če ni.

Klic `mozna_pot("ABCRVRIEIPNM", zemljevid)` vrne `True`, klic `mozna_pot("ABCRVREPNM", zemljevid)` pa `False`, ker ni povezave iz R v E.

Rešitev Bože, kaj se je dogajalo pri tej nalogi! :) Rešitev je preprosta: gremo čez pare zaporednih križišč na poti. Za vsak par pogledamo, ali nastopa (kot ključ) v slovarju, in čim zalotimo takšnega, ki ne, zatulimo "FALSCH!". Če preživimo do konca, pa `True`.

```
from itertools import pairwise
```

```

def mozna_pot(pot, zemljevid):
    for a, b in pairwise(pot):
        if (a, b) not in zemljevid:
            return False
    return True

```

Gre tudi malenkost krajše.

```
def mozna_pot(pot, zemljevid):
    for povezava in pairwise(pot):
        if povezava not in zemljevid:
            return False
    return True
```

Če rečemo, da morajo vse povezave nastopati kot ključi v slovarju, mora biti množica povezav podmnožica množice ključev, in rešitev je potemtakem lahko kar:

```
def mozna_pot(pot, zemljevid):
    return set(pairwise(pot)) <= set(zemljevid)
```

Prav ta teden pa se bomo učili, da se da ono različico z zanko napisati tudi učinkoviteje:

```
def mozna_pot(pot, zemljevid):
    return all(povezava in zemljevid for povezava in pairwise(pot))
```

Najbolj tipična komplikacija tule je bila, da niste uporabili `pairwise(pot)` (ali, po starem `zip(pot, pot[1:])`), temveč `range(len(pot))` in indekse. To je potem izgledalo tako:

```
def mozna_pot(pot, zemljevid):
    for i in range(len(pot) - 1):
        if (pot[i], pot[i + 1]) not in zemljevid:
            return False
    return True
```

To ni spodobno, je pa še OK. Veliko manj spodobne in manj OK so vse rešitve, ki iz kakega razloga delajo zanko čez `zemljevid`. To je počasno in ne izkorišča tega, kar nam nudijo slovarji. Slovarjev, nimamo zato, da bi delali zanke čeznje, temveč zato, da ne bi delali zank čeznje.

Zelo pogosto sem videval tudi rešitve, ki najprej zgradijo seznam povezav in gredo nato čezenj, recimo takole (ali pa, še bolj zapleteno, z `range` ali celo kaj daljšega):

```
def mozna_pot(pot, zemljevid):
    povezave = []
    for a, b in pairwise(pot):
        povezave.append((a, b))

    for povezava in povezave:
        if povezava not in zemljevid:
            return False
    return True
```

To je nepotrebno prelaganje podatkov med seznamami. S tem ne pridobimo ničesar, razen priložnosti za napake.

Naloga (nadaljevanje)

- Funkcija `potrebne_vescine(pot, zemljevid)` prejme enake argumente kot prejšnja funkcija, s tem da bo `pot`, ki jo prejme, bo vedno možna. Funkcija mora vrniti množico veščin, ki jih mora kolesar obvladati, če želi prevoziti to pot.

Klic `potrebne_vescine("RIPSTUT", zemljevid)` vrne `{'robnik', 'stopnice', 'makadam', 'trava'}`, saj so to veščine, ki jih potrebujemo za to pot (na zemljevidu označene kot *sr*, *g*, *rt* in *gt*).

Rešitev Naloga je malo podobna prejšnji, le da namesto preverjanja, ali je pot možna, sestavi množico in vanjo dodaja veščine, potrebne za vsako povezavo.

```
def potrebne_vescine(pot, zemljevid):
    vescine = set()
    for povezava in pairwise(pot):
        vescine |= zemljevid[povezava]
    return vescine
```

Jedrnatejša rešitev z generatorjem zahteva malo več znanja.

```
def potrebne_vescine(pot, zemljevid):
    return set().union(*(zemljevid[povezava] for povezava in pairwise(pot)))
```

Ta rešitev je primerno Pythonovska in razumljiva za veterana, tu pa je ne bomo razlagali.

Tudi to funkcijo lahko po nepotrebnem zapletemo na enake načine kot zgornjo. Pa še na enega zraven - in to je naredilo veliko študentov. Namesto

```
    vescine |= zemljevid[povezava]
```

so pisali

```
    for vescina in zemljevid[povezava]:
        vescine.add(povezava)
```

Če želimo v neko množico dodati vse elemente neke druge množice, je to pač unija.

Nekateri so bili še previdnejši in pred dodajanjem preverili, da ni element slučajno že v množici, `if zemljevid[povezava] not in vescine`. Tudi to je nepotrebno. Množica ne more dvakrat vsebovati istega elementa.

Nekatere je zmotilo, da njihove množice ne vsebujejo elementov v enakem vrstnem redu, v kakršnem so navedene v rešitvi. Kot sem opozoril na predavanju, vrstni red elementov v množici ni določen. Lahko se spreminja med izvajanjem. In vsakič, ko poženete program, je lahko drugačen. Množice nimajo koncepta "vrstnega reda", zato ta ni pomemben.

```
{1, 2, 3, 4} == {2, 1, 4, 3}
```

True

Ob tej nalogi se je dogajalo še nekaj zanimivega: nekateri so poskušali tole:

```
vescine.add(zemljevid[povezava])
```

To ne gre. To ni unija (unija doda v množico vse elemente neke druge množice), temveč dodajanje. Po tem, bi množica `vescine` vsebovala celo množico `povezava` kot *element*. Množice lahko vsebujejo samo nespremenljive stvari, množice pa so spremenljive. Zato množice ne morejo vsebovati množic. Pa četudi bi to šlo, ne bi bilo pravilno, saj mora funkcija vrniti množico veščin, ne pa množico množic veščin.

Naloga (nadaljevanje)

- Funkcija `nepotrebne_vescine(pot, zemljevid, vescine)` prejme enake argumente, poleg tega pa še množico veščin, ki jih obvlada kolesar. Vrniti mora množico veščin, ki so za to pot nepotrebne.

Klic `nepotrebne_vescine("IPNMNPO", zemljevid, {'stopnice', 'makadam', 'bolt', 'rodeo'})` vrne `{'stopnice', 'bolt'}`, saj tidve veščini za pot "IPNMNPO" nista potrebni.

Rešitev Kdor je napisal kaj več kot

```
def nepotrebne_vescine(pot, zemljevid, vescine):  
    return vescine - potrebne_vescine(pot, zemljevid)
```

ima preveč časa.

Naloga (nadaljevanje)

- Funkcija `kam(zemljevid, tocka, vescine)` prejme `zemljevid`, neko točko in množico veščin, ki jih obvlada nek kolesar. Vrniti mora množico točk, ki so neposredno povezane s podano točko in jih ta kolesar lahko doseže.

Klic `kam(zemljevid, R, {"stopnice", "trava", "pešci", "robnik", "avtocesta"})` vrne `{U, I, D}`. (Zmožnost vožnje po avtocesti je tu sicer nepotrebna, vendar nas ne ovira.)

Rešitev Tu bo potrebno iti čez celoten `zemljevid`, potrebovali pa bomo tako ključne kot vrednosti, torej gremo čez `zemljevid.items()`. Za vsako povezavo bomo preverili, ali je izhodiščna točka enaka podani in ali je množica potrebnih veščin podmnožica veščin, ki jih imajmo. Če je tako, dodamo cilj povezave med dosegljive točke.

```
def kam(zemljevid, tocka, vescine):  
    tocke = set()  
    for (od, do), potrebne in zemljevid.items():  
        if od == tocka and potrebne <= vescine:
```

```

        tocke.add(do)
    return tocke

```

Ko se naučimo o izpeljanih množicah, pa je rešitev krajša in jedrnatejša:

```

def kam(zemljevid, tocka, vescine):
    return {do
        for (od, do), potrebne in zemljevid.items()
        if od == tocka and potrebne <= vescine}

```

Naloga (nadaljevanje)

- Funkcija `dolgcas(zemljevid)` vrne množico povezav na podanem zemljevidu, ki ne zahtevajo nobene veščine. Vsak par naj vrne le enkrat - če je dolgačasna povezava med B in V, naj množica vsebuje bodisi (B, V) bodisi (V, B), ne pa obojega.

Klic `dolgcas(zemljevid)` lahko vrne, recimo, {(B, V), (S, P), (J, K)}, ali pa, na primer, {(K, J), (V, B), (S, P)}...

Rešitev Ta je dejansko malo dolgačasna. Gremo čez slovar in izbiramo dolgačasne povezave.

```

def dolgcas(zemljevid):
    dolgocasne = set()
    for (od, do), potrebne in zemljevid.items():
        if not potrebne and od < do:
            dolgocasne.add((od, do))
    return dolgocasne

```

Dolgačasno nalogo popestri to, da je potrebno vsako povezavo dodati le v eni smeri. Tu lahko izumljamo raketno znanost, lahko pa se odločimo, naj bo prva točka po abecedi vedno pred drugo - kot smo storili zgoraj. :)

Rešitev z izpeljano množico je takšna:

```

def dolgcas(zemljevid):
    return {(od, do)
        for (od, do), potrebne in zemljevid.items()
        if not potrebne and od < do}

```

Dodatna naloga

Napiši funkcijo `koncna_tocka(pot, zemljevid, vescine)`, ki prejme enake argumente kot `nepotrebne_vescine`. Vrniti mora dve stvari: točko, do katere lahko kolesar s temi veščinami prevozi to pot, in množico veščin, ki mu manjkajo, da bi šel naprej iz te točke. Ta množica naj ne vključuje morebitnih drugih veščin, ki bi ga čakale na nadaljnji poti, temveč le manjkajoče veščine za prvo povezavo, ki je ne uspe prevoziti.

Klic `koncna_tocka("ABCRVB", zemljevid, {"makadam", "trava"})` vrne `("B", {'lonci', 'bolt'})`. Kolesar, ki bi se namenil iti po poti "ABCRVB" bi se zataknil v točki B, ker ne zna voziti slaloma med cvetličnimi lonci in odvrženimi skiroji. Nadaljnja pot bi sicer zahtevala še druge veščine (recimo spust po stopnicah med C in R), vendar funkcija ne gleda naprej.

Rešitev

Gremo po poti. Če naletimo na povezavo, ki zahteva veščine, ki jih kolesar ne premore, vrne trenutno točko in množico veščin, potrebnih za to povezavo, ki ji odštejemo množico veščin, ki jih kolesar obvlada. Če pridemo do konca, pa vrnemo zadnjo točko in prazno množico.

```
def koncna_tocka(pot, zemljevid, vescine):
    for a, b in pairwise(pot):
        if not zemljevid[(a, b)] <= vescine:
            return a, zemljevid[(a, b)] - vescine
    return pot[-1], set()
```

Zanimiv je pogoj, `not zemljevid[(a, b)] <= vescine`. To ni isto kot `zemljevid[(a, b)] > vescine`. Pogoj `not zemljevid[(a, b)] <= vescine` pravi, da *ni res, da množica `vescine` vsebuje vse veščine, ki so potrebne na povezavi (in morda še kakšno zraven)*. Pogoj `zemljevid[(a, b)] > vescine` pa pravi, da *povezava zahteva vse veščine, ki jih premore kolesar in še vsaj eno zraven*.

Recimo, da imamo

```
vescine = {"a", "b", "c"}
na_povezavi = {"c", "d"}
```

Pogoj, ki preveri, ali se kolesar tu ustavi, se glasi

```
not na_povezavi <= vescine
```

`True`

`True`, saj se kolesar dejansko ustavi, ker ne obvlada veščine d.

Pogoj, ki je navidez le poenostavljena oblika gornjega, pa ne vrne istega, pravega rezultata.

```
na_povezavi > vescine
```

`False`

Če sta `x` in `y` dve števili, velja, da je `not x <= y` isto kot `x > y`. Če sta `x` in `y` množici, pa pač ni tako.

Reševanje te naloge z generatorji je bolj zabavno (in ne sodi ravno v obvezno znanje za izpit! :). Sam sem razmišljal takole: grem po poti, dokler ne pridem do

povezave, ki je ne morem prevoziti. Vrnem prvo točko te povezave in manjkajoče veščine.

```
from itertools import dropwhile
```

```
def koncna_tocka(pot, zemljevid, vescine):  
    pov = next(dropwhile(lambda pov: pov in zemljevid and zemljevid[pov] <= vescine, pairwise(pot)))  
    return pov[0], zemljevid[pov] - vescine
```

Funkcija `dropwhile` prejme predikat (funkcijo, ki vrača `bool`) in neko zaporedje. `dropwhile` vrne vse elemente od prvega, za katerega predikat vrne `True`. Zanima nas le prvi od teh; dobimo ga z `next`.

Gornja funkcija deluje pravilno samo, če poti ni možno prevoziti do konca. Če jo je in moramo vrniti zadnjo točko in prazno množico, se reč še malo zaplete.

Med predavanjem je eden od študentov sestavil veliko preprostejšo rešitev od moje. Pa še pravilno deluje tudi, če je pot možno prevoziti v celoti.

```
def koncna_tocka(pot, zemljevid, vescine):  
    return next(((povezava[0], zemljevid[povezava] - vescine)  
                for povezava in pairwise(pot)  
                if povezava not in zemljevid or not zemljevid[povezava] <= vescine),  
                (pot[-1], set()))
```

Ta funkcija generira vse neprevozne povezave (točneje: prvo točko in manjkajoče veščine), vendar z `next` vrne le prvo. Če so vse povezave možne, pa kot privzeto vrednost vrne zadnjo točko na poti in prazno množico.

Moje kompliciranje kaže, kako hitro se človek zagozdi v napačno razmišljanje. Študentska rešitev pa je res elegantna. Kudos.