

Na MOL-ovem Oddelku za motorni promet in gospodarske dejavnosti na vsako opozorilo o neprijetnostih na kolesarskih poteh odgovorijo, da bodo *preverili in ukrepali skladno s svojimi pristojnostmi*. Še več, njihovi uslužbenci vsakodnevno, praktično 24/7, kolesarijo po Ljubljani in identificirajo nevarne točke, da jih čimprej odpravijo. Funkcije, ki jih boste napisali tu, jim bodo pomagale še boljše in vestnejše opravljati njihovo delo.

Da bo naloga lažja, bomo predpostavili, da ima mesto pravokotno mrežo.

Zemljevidi so zapisani v datotekah s takšno vsebino:

```
.r.....c..c
r.b.....
.p...r.....
.....
.s.....l....
.ad.....a..
.....
..gl.....g.
```

Za lažje sledenje navodilom, je tule zemljevid z oštevilčenimi stolpci in vrsticami. V datoteki teh števil ni.

```

11
012345678901
0 .r.....c..c
1 r.b.....
2 .p...r.....
3 .....
4 .s.....l....
5 .ad.....a..
6 .....
7 ..gl.....g.
```

- Koordinate zapisujemo tako, da navedemo številko stolpca, nato številko vrstice. Štejemo od 0. Koordinata polja s črko **b** je (2, 1).
- Razdalje merimo tako, da štejemo koliko vodoravnih in navpičnih korakov je potrebno narediti med črkama. Razdalja med **a** in **g** je 3, ker moramo en korak desno in dva dol. Razdalja med **p** in **b** je 2.
- Katere črke se pojavljajo v zemljevidu, ne vemo vnaprej. V nekem mestu kolesarji morda vozijo po poteh, sestavljenih iz `$#!?@@!##!`. Vemo le, da . pomeni, da na polju ni ničesar posebnega.

Ocena 6

1. preberi_zemljevid(ime_dat)

Napiši funkcijo `preberi_zemljevid(ime_dat)`, ki dobi ime datoteke z zemljevidom in vrne zemljevid, shranjen na poljuben način, ki se ti zdi smiseln. Vse

nadaljnje funkcije, ki jih bo potrebno napisati, bodo dobile zemljevid v takšni obliki, kot ga boš sestavil v tej funkciji.

Nasvet: preberi ostale naloge in razmisli, kako si shraniti zemljevid, da ti bo z njim udobno delati.

Funkcija lahko vrne, karkoli želiš - seznam, slovar, ime datoteke, množico, seznam nizov ... ali celo terko, v kateri sta slovar in niz. Testi bodo preverjali samo, da funkcija vrne nekaj, kar ni `None`, zato sam skrbno preveri, da funkcija res vrača, kar si si zamislil, naj vrne.

Rešitev V naslednjih nalogah - pravzaprav že kar v naslednji funkciji - bomo pogosto iskali najbližji tak in tak znak, redkeje pa nas ne bo zanimalo, kaj je na nekih koordinatah. Torej je smiselno zemljevid organizirati okrog znakov, ne okrog koordinat. Naša funkcija `preberi_zemljevid` bo zato vrnila slovar, katerega ključi bodo znaki, pripadajoče vrednosti pa množice koordinat, na katerih se ta znak pojavi.

```
from collections import defaultdict

def preberi_zemljevid(ime_dat):
    lokacije = defaultdict(set)
    for v, vrstica in enumerate(open(ime_dat)):
        for s, c in enumerate(vrstica):
            if c not in ".\n":
                lokacije[c].add((s, v))
    return lokacije
```

Čez datoteko gremo s `for v, vrstica in enumerate(open(ime_dat))`. Poleg vsebine vrstice bomo potrebovali tudi njeno številko, da bomo lahko zapisovali koordinate.

Tudi čez vrstico gremo z `enumerate`: `s` bo stolpec, `c` (kot *character*) pa znak.

Zanimajo nas le znaki, ki niso pike, mimogrede pa se znebimo še `\n`. V `lokacije[c]`, ki beleži lokacije znakov `c`, dodamo par `s` stolpcem in vrstico. Ker so `lokacije` `defaultdict(set)` in ne običajen slovar, se bodo novi ključi pojavljali sami od sebe, kadar bo potrebno.

2. najblizji(x, y, c, zemljevid)

Napiši funkcijo `najblizji(x, y, c, zemljevid)`, ki prejme

- koordinate neke točke `(x, y)`,
- iskani znak ali prazen niz `(c)`,
- in zemljevid v obliki, v kateri ga vrača tvoja funkcija `preberi_zemljevid(zemljevid)`.

Delovanje funkcije je odvisno od argumenta `c`:

- Če `c` vsebuje znak, mora funkcija vrniti koordinate polja, ki vsebuje črko `c` in je najbližje podanemu polju (vendar **ni enako podanemu** polju!), ki vsebuje črko `c`. Klic `najblizji(8, 2, "r", zemljevid)` vrne `(5, 2)`. Klic `najblizji(1, 5, "a", zemljevid)` vrne `(9, 5)`; `a` je sicer tudi ravno na točki `(1, 5)`, vendar funkcija ne sme vrniti iste točke.
- Če je `c` prazen niz, funkcija vrne koordinato najbližjega polja (ki ni trenutno polje) in vsebuje poljuben znak (razen, seveda pike).

Kadar je enako oddaljenih koordinat več, vrne tisto v bolj levem stolpcu. Če je tudi teh več (`= 2`, se mi zdi :), vrne tisto, ki je višje.

Če iskana črka ne obstaja, funkcija vrne `None`.

Rešitev Začnimo z dolgo rešitvijo, potem pa jo bomo skrajšali.

```
def najblizji(x, y, c, zemljevid):
    if c:
        kandidati = zemljevid[c]
    else:
        kandidati = set()
        for koordinata in zemljevid.values():
            kandidati |= koordinata

    najkoord, najrazdalja = None, None
    for koord in kandidati:
        razdalja = abs(koord[0] - x) + abs(koord[1] - y)
        if koord != (x, y) and (
            najrazdalja is None
            or razdalja < najrazdalja
            or razdalja == najrazdalja and koord < najkoord):
            najkoord, najrazdalja = koord, razdalja
    return najkoord
```

Prvi del funkcije sestavi seznam koordinat, ki vsebujejo iskani znak ali pa, če ta ni podan, katerikoli znak. Če je `c` neprazen niz, je stvar preprosta: koordinate najdemo v `zemljevid[c]`. Sicer pa sestavimo prazno množico in vanjo priunijamo vse, kar najdemo v `zemljevid.values()`.

Sledi klasična zanka, s katero iščemo nek minimum prek nečesa. Tokrat gremo prek koordinat, za vsako izračunamo razdaljo do koordinat, podanih kot argument. Trenutne koordinate so boljše od najboljših, če niso enake podanim in bodisi nismo videli še nobenih (`najrazdalja is None`) bodisi je ta razdalja krajša od najkrajše doslej, bodisi ji je enaka, vendar so koordinate "manjše".

Če iskanega znaka ni, se zanka ne bo izvedla nikoli, `najkoord` bo ostal `None` in funkcija bo vrnila `None`, kot hočejo navodila.

Zdaj pa krajša različica. Iz rokava, točneje, iz modula `itertools` privlečemo

funkcijo `chain`. Tej lahko podamo seznam nekih stvari, pa jih bo spela skupaj.

```
from itertools import chain

for x in chain("ab", [1, 3, 4]):
    print(x)
```

```
a
b
1
3
4
```

Naslednji trik, ki ga najbrž že poznamo, je razpakiranje argumentov. Če imamo, recimo, nek seznam ali karkoli, čez kar je možno iterirati, in bi radi njegove elemente uporabili kot argumente funkcije, ga podamo kot argument tako, da predenj dodamo zvezdico.

```
def f(a, b, c, d, e):
    print(a, b, c, d, e)
```

```
s = [1, 2, 3]
f(0, *s, 4)
```

```
s = "ab"
f(0, 1, 2, *s)
```

```
0 1 2 3 4
0 1 2 a b
```

Tako lahko celoten prvi del funkcije nadomestimo z

```
kandidati = zemljevid[c] if c else chain(*zemljevid.values())
```

Sledi drugi del. Iščemo minimum prek vseh kandidatov, razen tistega, ki je enak podanim koordinatam, `for k in kandidati if k != (x, y)`. Koordinate primerjamo glede na razdalje od izbrane; če bo razdalja enaka, pa hočemo primerjati koordinate same. Ključ za primerjavo bo torej `(abs(k[0] - x) + abs(k[1] - y), k)` - najprej razdalja, če je ta enaka, koordinate. Končno, če kandidatov ni, mora `min` vrniti `None`; to mu povemo z argumentom `default`. Skrajšana funkcija je

```
def najblizji(x, y, c, zemljevid):
    kandidati = zemljevid[c] if c else chain(*zemljevid.values())
    return min((k for k in kandidati if k != (x, y)),
               key=lambda k: (abs(k[0] - x) + abs(k[1] - y), k),
               default=None)
```

3. najpogostejsi(x, y, d, zemljevid)

Napiši funkcijo `najpogostejsi(x, y, d, zemljevid)`, ki vrne najpogostejšega med znaki, ki so za največ `d` korakov oddaljen od trenutnega polja. Upoštevaj tudi trenutno polje. Če je takih znakov več (če so na razdalji `d`, recimo po trije znaki `b`, `l` in `r`), naj vrne tistega, ki je prej po abecedi (v tem primeru `"b"`). Če na tej razdalji ni nobenega znaka, vrne `None`.

Rešitev Poškilimo naprej in vidimo, da bomo pisali funkcijo, ki vrne množico vseh najpogostejših znakov. Ta naloga pravi, da moramo vrniti tistega med njimi, ki je prvi po abecedi. Poiskal nam ga bo `min`. Povedati mu moramo le še, kaj naj vrne, če je množica prazna - `None`.

```
def najpogostejsi(x, y, d, zemljevid):
    return min(vsi_najpogostejsi(x, y, d, zemljevid),
               default=None)
```

4. vsi_najpogostejsi(x, y, d, zemljevid)

Napiši funkcijo `vsi_najpogostejsi(x, y, d, zemljevid)`, ki je podobna prejšnji funkciji, vendar vrne množico najpogostejših črk. V gornjem primeru torej namesto `"b"` vrne `{"b", "l", "r"}`. Če je najpogostejši le eden, vrne množico z enim elementom. Če ni nobenega, vrne prazno množico.

Rešitev Na tem mestu podvomimo, ali je bila naša odločitev, da bo `zemljevid` slovar znakov in ne slovar koordinat, pravilna. Stisnimo zobe in sprogramirajmo. Spet najprej na dolgo.

```
def vsi_najpogostejsi(x, y, d, zemljevid):
    pojavitve = defaultdict(int)
    for c, coords in zemljevid.items():
        for x0, y0 in coords:
            if abs(x0 - x) + abs(y0 - y) <= d:
                pojavitve[c] += 1

    maxpojav = max(pojavitve.values(), default=0)
    pogosti = set()
    for c, pojavitev in pojavitve.items():
        if pojavitev == maxpojav:
            pogosti.add(c)
    return pogosti
```

V prvem delu gremo čez vse znake in znotraj tega čez vse koordinate. Število pojavitev beležimo v `pojavitve`. Če je znak znotraj zahtevane razdalje, k številu njegovih pojavitev prištejemo 1.

V drugem delu ugotovimo največje število pojavitev. Če na podani razdalji ni nobene ovire (v Ljubljani je to seveda možno, le razdalja mora biti zelo majhna),

naj bo največje število pojavitev enako 0 - lahko pa bi bilo karkoli, saj se zanka, ki sledi in v kateri dejansko uporabimo to spremenljivko, tako ali tako ne bo izvedla nikoli.

Nato gremo čez preštete pojavitve in v množico zložimo vse znake, ki se pojavijo tolikokrat, kolikorkrat se pojavi najpogostejši.

Nekoliko daljša, a morda učinkovitejša različica drugega dela je

```
maxpojav = 0
pogosti = set()
for c, pojavitev in pojavitve.items():
    if pojavitev > maxpojav:
        maxpojav = pojavitev
        pogosti = {c}
    elif pojavitev == maxpojav:
        pogosti.add(c)
```

Tu največjega števila pojavitev ne ugotovimo pred zanko, temveč kar v zanki sestavljamo množico z znaki, ki imajo največje število pojavitev doslej. Če se pojavi pogostejši znak, začnemo množico sestavljati od začetka.

V Pythonu takšna rešitev najbrž ni bistveno hitrejša od prve, saj je iskanje maksimuma s funkcijo `max` hitra operacija. V kakem hitrejšem jeziku in pri večjem številu podatkov pa bi utegnila biti druga različica hitrejša.

Zdaj pa še krajša, bolj Pythonovska rešitev.

```
def vsi_najpogostejši(x, y, d, zemljevid):
    pojavitve = {c: sum((abs(x0 - x) + abs(y0 - y) <= d
                        for x0, y0 in koords))
                 for c, koords in zemljevid.items()}
    maxpojav = max(pojavitve.values(), default=0)
    return {k for k, v in pojavitve.items() if v == maxpojav > 0}
```

Slovar `pojavitve` sestavimo z generatorjem. Generator bo seveda tekel prek `zemljevid`. Ključi `zemljevida` bodo tudi ključi slovarja `pojavitve`, pripadajoče vrednosti pa bodo število pojavitev znaka. Te dobimo tako, da za vsake koordinate ugotovimo ali so znotraj podane razdalje; `True`-je preštejemo s `sum`.

Nato, tako kot prej, ugotovimo, kolikokrat se pojavi najpogostejši znak. Ker bodo `pojavitve` vsi ključi `zemljevida`, je `default=0` tule skoraj nepotreben, saj bomo v takšni situaciji le, če na celem `zemljevidu` ni niti ene ovire, to pa je, vemo, realno nemogoče.

Nato vrnemo množico, v katero zložimo vse ključe slovarja `pojavitve` (= vse znake), za katere je pripadajoča vrednost (= število pojavitev) enaka maksimalni. Zraven pa zvito dodamo pogoj, da mora biti `maxpojav > 0`. Če ni, to pomeni, da znotraj podane razdalje ni bilo nobenega znaka in v tem primeru bo množica prazna.

Ocena 7

1. najblizji(x, y, c, zemljevid, prepovedani)

Spremeni obstoječo funkcijo `najblizji` tako, da bo sprejemala dodatni argument, množico `prepovedani`. Funkcija `najblizji(x, y, c, zemljevid, prepovedani)` naj vrne enak rezultat kot `najblizji`, vendar ignorira polja, naštet v `prepovedani`.

(Ne skrbi za teste. To ni tvoj problem. Samo dodaj argument. :)

Rešitev Da ne bi ponovno pisali dolge različice funkcije, le povejmo, kaj moramo spremeniti: v glavo funkcije dodamo zahtevani argument `prepovedani`, pogoj

```
if koord != (x, y) and (
```

pa dopolnimo v

```
if koord != (x, y) and koord not in prepovedani and (
```

Podobno dopolnimo krajšo različico, ki se spremeni v

```
def najblizji(x, y, c, zemljevid, prepovedani):
    kandidati = zemljevid[c] if c else chain(*zemljevid.values())
    return min((k for k in kandidati if k != (x, y) and k not in prepovedani),
               key=lambda k: (abs(k[0] - x) + abs(k[1] - y), k),
               default=None)
```

2.. angelca(x, y, znamenitosti, zemljevid)

Angelca meni, da je potrebno stvari delati sistematično. (To vedno ponavlja tudi županu!) Zato za vsakega nadzornika pripravi dnevni načrt: začetno točko (ta je vedno "prazna", `.`) in spisek mestnih kolesarskih znamenitosti (robniki, stopnice, bolt ipd).

Nadzornik opravlja vožnjo tako, da obišče znamenitosti v vrstnem redu, ki ga je določila Ang

Recimo, da začne kar na `(0, 0)` in mora prevoziti spisek ``rprbr``.

- Najprej gre na ``r`` na koordinatah `(0, 1)`. Pot je dolga ``1``.
- Nato gre na ``p`` na `(1, 2)`. To zahteva ``2`` koraka.
- Nadaljuje na ``r`` na `(1, 0)` (na prejšnjem ``r`` je bil, torej se ne vrača tja). Tudi to je
- Nato gre na ``b`` na `(2, 1)`. Spet ``2`` koraka.
- Konča na ``r`` na `(5, 2)` (dva bližja ``r``-ja je že obiskal). To je dolgo ``4`` korake.

Skupno je naredil $1 + 2 + 2 + 2 + 4 = 11$ korakov.

Nek drug nadzornik začne na `(0, 5)` in mora obiskati ``asalag``.

- Gre na `a` na `(1, 5)` : `1` korak.
- Gre na `s` na `(1, 4)` : `1` korak.
- Gre na `a` na `(9, 5)` : `9` korakov.
- Gre na `l` na `(7, 4)` : `3` koraki.
- Tu bi moral obiskati `a`, vendar se ustavi, ker ni več nobenega `a`-ja, ki ga še ni obiskal.

Skupno je naredil $1 + 1 + 9 + 3 = 14$ korakov.

Napiši funkcijo `angelca(x, y, znamenitosti, zemljevid)`, ki vrne število korakov, ki jih bo naredil.

Rešitev Funkcija je dokaj "mehanska" - nobenih eksotičnih podatkovnih struktur, trikov in posebej "pythonovskih" rešitev. Korakoma korakamo po korakih, seštevamo razdalje in prepovedujemo polja, ki jih obiščemo.

```
def angelca(x, y, znamenitosti, zemljevid):
    prepovedani = set()
    korakov = 0
    for znamenitost in znamenitosti:
        nasl = najblizji(x, y, znamenitost, zemljevid, prepovedani)
        if not nasl:
            break
        xn, yn = nasl
        korakov += abs(xn - x) + abs(yn - y)
        x, y = xn, yn
        prepovedani.add((x, y))
    return korakov
```

Ocena 8

Ko je Angelca na dopustu, jo zamenja Johanca. Ta deluje drugače. Nadzornik dobi začetno točko (na kateri ni ničesar, le `.`) in niz z opisom premikov.

Na primer, da začne na `(1, 1)` in mora slediti poti `><>4v^12^13v1<2^>>>>>2v^4v4<^`

- `>`: Najprej gre desno. Vidi `b`.
- `<`: Gre levo. Tam ni ničesar.
- `>`: Gre desno. Tam je `b`, vendar ga je že videl, zato ga ne zabeleži.
- `4v`: Nato gre za 4 dol. Vidi `d`.
- `^`: Gre gor. Tam ne vidi ničesar.
- `12^`: Gre še 12-krat gor. Tam ne vidi sploh ničesar, saj tega niti ni na zemljevidu.
- `13v`: Gre 13-krat dol, torej je spet tam, kjer je bil `d`. Ker je ta `d` že videl, ne šteje.
- `1<`: Gre za 1 levo. Tam vidi `a`.
- `2^`: Gre za 2 gor. Tam ni ničesar. **s, ki ga je preskočil, spregleda!**
- `>>>>>`: Nato gre šestkrat `>`. Ničesar.
- `2v`: Gre za 2 dol, pri čemer preskoči 1.

- `^` Gre gor in vidi 1, ki ga je prej preskočil.
- `^v`: Gre dol in gor. Ponovno vidi 1, vendar ga ne zabeleži, ker je tam že bil.
- `4v4<^`: v nadaljevanju se spusti pod zemljevid, gre levo in ko se vrne gor, naleti na (drug) 1 ter ga zabeleži.

Napiši funkcijo, `johanca(x, y, pot, zemljevid)`. Funkcija mora vrniti niz z zaporedjem znamenitosti, ki jih je videl kolesar. V gornjem primeru vrne `bdall`.

Pot je, kot kaže gornji primer, sestavljena iz zaporedja znakov `v, ^, <` in `>`.

- Če je pred znakom številka (ki ima lahko poljubno število mest), to pomeni, da naredi več korakov v tisto smer. Ker brezglavo divja, ne vidi znamenitosti, ki so vmes. Tako je, recimo, pri `"*Gre za 2 gor`.
- Če pot vodi ven iz zemljevida, se pač vozi tam - le da tam pač ne vidi ničesar.
- Ko vidi nekaj, kar je že videl (kot recimo `d` v gornjem primeru), to ignoriramo.
- Gornje velja le za znamenitosti na *istih* koordinatah. Ko v gornjem primeru naleti na nov 1, ga zabeleži.

Rešitev Tegoba te naloge je, kajpak, v tem, da moramo brati zoprni niz. Osnovna ideja bo takšna: gremo čez niz. Vsakič, ko naletimo na števko, jo dodamo v niz `stevilka`. Ko naletimo na smer, se premaknemo za toliko, kolikor piše v `stevilka`, ali za 1, če je `stevilka` prazen niz. Po premiku nastavimo `stevilka` nazaj na prazen niz.

```
def johanca(x, y, pot, zemljevid):
    ...
    stevilka = ""
    for c in pot:
        if c in "<>^v":
            # premakni se za `stevilka` (ali 1, če je `stevilka` prazna) polj v smer `c`
            ...

            stevilka = ""
        else:
            stevilka += c
    ...
```

K temu okostju dodamo še detajle. `znamenitosti` bo niz z obiskanimi znamenitostmi, ki ga bomo vrnili na koncu funkcije. `obiskane` bo množica obiskanih koordinat; potrebna je, da preprečimo, da bi večkrat zabeležili isto znamenitost.

```
def johanca(x, y, pot, zemljevid):
    znamenitosti = ""
    obiskane = set()
    stevilka = ""
```

```

for c in pot:
    if c in "<>^v":
        stevilka = int(stevilka or 1)
        x, y = {"<": (x - stevilka, y), ">": (x + stevilka, y), "v": (x, y + stevilka), "^": (x, y - stevilka)}[c]
        if (x, y) not in obiskane:
            obiskane.add((x, y))
            for z, koord in zemljevid.items():
                if (x, y) in koord:
                    znamenitosti += z
        stevilka = ""
    else:
        stevilka += c
return znamenitosti

```

Izraz `stevilka or 1` bo imel vrednost `stevilka`, če je niz `stevilka` resničen, torej neprazen, in `1`, če je neresničen, prazen. To pretvorimo v `int`, pa imamo razdaljo.

Nove koordinate bi lahko dobili z zaporedjem `if c == "<"` in tako naprej. Ker smo leni, raje sestavimo slovar, ki za vsak možen premik vsebuje nove koordinate, potem pa s `[c]` izberemo ustrezne.

Po premiku preverimo, ali smo na tem polju že bili. Če nismo, dodamo polje med obiskana polja in po zemljevidu pobrskamo, ali polje vsebuje kakšno znamenitost.

Ocena 9

- Angelca ima sestrično v nekem vlemestu in ta ima podobno velik vpliv na tamkajšnjega župana. Zato moraš poskrbeti, da bo funkcija `angelca` delovala tudi na bistveno večjem zemljevidu. Testi bodo preverjali, da se funkcija izvede v doglednem času - biti mora dovolj hitra, da preživi teste. Če si jo že prej napisal učinkovito, ti ni potrebno storiti ničesar. Če nisi - razmisli in popravi.
- Johanca tudi. :)
- Ker želijo tudi v teh, velikih mestih, pritegniti kolesarje k ogledu kolesarskih znamenitosti, napiši še funkcijo `najboljsa_cetrta(a)`. Ta poišče kvadrat s stranico `a`, ki vsebuje največ znamenitosti. Funkcija vrne koordinate njegovega zgornjega levega oglišča. Tudi ta funkcija se mora končati v doglednem času.

Če je enako dobrih kvadratov več, vrne najbolj levega, med enako dobrimi najbolj levimi pa najbolj zgornjega.

Rešitev Ob prvih dveh smo hvaležni sami sebi, da smo zemljevid zapisali na način, na katerega smo ga zapisali. Ob tretji se za to odločitev nase razjemo, zavijamo rokave in programiramo. Ker gre tu vendarle za oceno 9, bomo kar takoj pokazali zaresno rešitev, ne kakšnih neskončnih zank. (Pač pa se bo,

spoiler, pokazalo, da je ta rešitev za zemljevide s pregosto posejanimi ovirami lahko počasna.)

```
def najboljsa_cetrt1(a, zemljevid):
    tocke = set().union(*zemljevid.values())
    maxx = max(x for x, _ in tocke)
    maxy = max(y for _, y in tocke)
    maxn = 0
    for x in range(max(1, maxx - a + 2)):
        for y in range(max(1, maxy - a + 2)):
            n = sum(x <= tx < x + a and y <= ty < y + a for tx, ty in tocke)
            if n > maxn:
                maxn = n
                xx, yy = x, y
    return xx, yy
```

Najprej sestavimo množico s koordinatami vseh znamenitosti. Množica, `set` ima metodo `union`, ki vrne unijo te množice in vseh množic, ki jih podamo kot argument. Sestavimo torej prazno množico in kot argument podamo vse množice v zemljevidu - zvezdice poskrbi, da argument ne bo en sam, namreč, `zemljevid.values()`, temveč bo argumentov več, namreč, argumenti bodo vsi elementi `zemljevid.values()`.

Nato poiščemo velikost zemljevida, največjo koordinato točke + 1, ker štejemo od 0.

Zdaj gremo prek vseh možnih levih in gornjih vogalov. Z vsakega gremo prek vseh točk (`for tx, ty in tocke`) in preštejemo, koliko od teh točk se nahaja znotraj kvadrata z gornjim levim ogliščem v (x, y) . Če jih je več kot največ doslej, bingo.

Tule je malo skrajšana rešitev - vedno mi gre na živce, ko moram ročno programirati `max`.

```
def najboljsa_cetrt(a, zemljevid):
    tocke = set().union(*zemljevid.values())
    maxx = max(x for x, _ in tocke) + 1
    maxy = max(y for _, y in tocke) + 1
    return max(((x, y)
                for x in range(max(1, maxx - a + 1)) for y in range(max(1, maxy - a + 1))),
               key=lambda k: sum(k[0] <= tx < k[0] + a and k[1] <= ty < k[1] + a for tx, ty
```

Preštevanje znamenitosti znotraj kvadrata se da tudi obrniti. Namesto da gremo čez seznam točk in preverjamo ali so znotraj kvadrata, lahko sestavimo množico točk v kvadratu in preverimo velikost njenega preseka z množico znamenitih točk.

```
def najboljsa_cetrt(a, zemljevid):
    tocke = set().union(*zemljevid.values())
    maxx = max(x for x, _ in tocke) + 1
```

```

maxy = max(y for _, y in tocke) + 1
maxn = 0
for x in range(max(1, maxx - a + 1)):
    for y in range(max(1, maxy - a + 1)):
        n = len({(tx, ty) for tx in range(x, x + a) for ty in range(y, y + a)}
                & tocke)
        if n > maxn:
            maxn = n
            xx, yy = x, y
return xx, yy

```

Ali, različica, v kateri uporabimo max:

```

def najboljsa_cetrt(a, zemljevid):
    tocke = set().union(*zemljevid.values())
    maxx = max(x for x, _ in tocke) + 1
    maxy = max(y for _, y in tocke) + 1
    return max(((x, y)
                for x in range(max(1, maxx - a + 1)) for y in range(max(1, maxy - a + 1))),
               key=lambda k: len({(tx, ty) for tx in range(k[0], k[0] + a) for ty in range(
                    & tocke)}))

```

Ker Python hitro sestavlja množice in računa njene preseke, je ta rešitev hitrejša od gornje - predvsem za mesta z veliko znamenitostmi.

Ob sestavljanju naloge sem imel v mislih drugačno rešitev: rešitev, v kateri izračunamo le število znamenitosti v kvadratu na levem robu, nato pa s tem kvadratom drsimo proti desni in vsakič odštejemo znamenitosti, ki smo jih izgubili na levi strani ter prištejemo znamenitosti na desni. Če so kvadrati res veliki, bo sprejemljivo hiter le takšen postopek. V tej nalogi se je izkazalo, da kljub velikim zemljevidom to ni potrebno.

Ocena 10

Kljub veliki skrbi za popisovanje nevarnih točk, nekatere vseeno ostajajo. Problem se utegne skrivati tule. Angelco in Johanco so dali za nekaj časa izpolnjevati stolpce v Excelu in nadzorniki se vozijo po drugačnem postopku.

Prvi del

Vsak nadzornik ima pri sebi n obrazcev. Pot začne v neki predpisani točki (x, y) , ki je gotovo prazna. V vsakem koraku si izbere neko znamenitost, ki je oddaljena največ d . Popiše jo. Nato izbere naslednjo točko, oddaljeno d . Popiše in gre do naslednje. To počne toliko časa, dokler mu ne zmanjka obrazcev. K znamenitostim, ki jih je že popisal, se nikoli ne vrača.

Napiši funkcijo `dosegljive(x, y, d, n, zemljevid)`, ki vrne množico točk, ki jih lahko doseže nadzornik. To ne pomeni, da lahko nadzornik obišče vse točke, temveč, da lahko obišče *vsako*.

Rešitev Funkcija `dosegljive` bo sestavila množico vseh točk in poklicala funkcijo `doseg`, ki bo imela enake argumente kot `dosegljive` in še enega dodatnega: točke, v katere je nadzorniku dovoljeno iti. V začetku bodo to kar vse točke.

`doseg` bo rekurzivna funkcija, ki bo v vsakem koraku preverila, ali ima nadzornik še kak obrazec ter v tem primeru dodala vse točke, ki so dosegljive iz trenutne točke, ter (zdaj pride rekurzije) vse točke, ki so dostopne iz teh, dosegljivih točk, če ima nadzornik en obrazec manj in če je ta točka zdaj prepovedana.

```
def dosegljive(x, y, d, n, zemljevid):
    def doseg(x, y, obr, dovoljene):
        dos = set()
        if obr:
            for xn, yn in dovoljene:
                if 0 < abs(xn - x) + abs(yn - y) <= d:
                    dos.add((xn, yn))
                    dos |= doseg(xn, yn, obr - 1, dovoljene - {(xn, yn)})
        return dos

    tocke = set().union(*zemljevid.values())
    return doseg(x, y, n, tocke)
```

Drugi del

Na točkah, označenih z *, so MOL-ove izpostave. Če nadzornik obišče izpostavo, tam dobi nove obrazce, tako da jih ima spet `n`.

- Tudi izpostave, ki jih obišče, smejo biti od trenutne pozicije oddaljene največ `d`.
- Isto izpostavo sme obiskati večkrat.
- Izpostave ne sme obiskati, če ima še vedno pri sebi `n` obrazcev. Kot pravi Angelca, to ne bi imelo nobenega smisla.

Dopolni funkcijo `dosegljive`, da bo upoštevala še ta pravila.

Nadzornik, ki nadzira spodnji zemljevid in začne pot na (11, 5), dela korake dolžine do 3 ter ima pri sebi največ 2 obrazca, lahko obišče {(7, 4), (11, 0), (10, 7), (8, 0), (9, 5)}.

```

      11
012345678901
0 .r..*...c..c
1 r.b.....
2 .p...r...*..
3 .*.....
4 .s.....l...
5 .ad.....a..
6 ..*.....
```

7 ..g1.....g.

Predlog: nalogo je možno rešiti rekurzivno ali iterativno. Rekurzivna rešitev bo najbrž preprostejša in bližja temu, kar smo se učili. Čez teden bom bistveno dopolnil zapiske o rekurziji. V drugem delu bo rekurzija eksplodirala, če je ne boste ukrotili s trikom, ki sem ga pokazal na predavanju (in bo seveda tudi v zapiskih).

Če se boste naloge lotili rekurzivno: funkcija `dosegljive` skoraj gotovo ne bo rekurzivna, pač pa bo poklicala, drugo, rekurzivno funkcijo, katere ime in argumente si zamislite sami.

Vzpodbuda: če razumete rekurzijo in se je znate lepo lotiti, je naloga za oceno 10 lahka in prijetna, rešitev pa kratka in elegantna. Ni pa ravno čisto trivialna.

Rešitev Gornjo funkcijo bo potrebno malo obdelati. :)

- V izpostave shranimo vse izpostave, `izpostave = zemljevid["*"]`. Ker je funkcija `doseg` znotraj funkcije `dosegljive`, ni potrebno, da bi bile izpostave argument funkcije `doseg`.
- Med dovoljenimi točkami ni več tistih, ki vsebujejo zvezdice. Roko-hitrski način, da jih poiščemo, je `tocke = set().union(*(v for k, v in zemljevid.items() if k != "*"))`. Če ne znate tako, je očitno možno tudi z navadno zanko.
- Znotraj funkcije bomo imeli poleg rekurzivnih klicev na nove znamenitosti (če ima nadzornik še kak `obr`) tudi rekurzivne klice na izpostave (če je nadzornik od zadnjega obiska na izpostavi porabil vsaj en obrazec, `obr < n`). Tako kot točka mora biti tudi izpostava dovolj blizu, rekurzivni klic pa je drugačen: namesto, da bi imel nadzornik po obisku izpostave `obr - 1` obrazcev, jih ima `n`. Poleg tega ostanejo dovoljene točke seveda enake.
- Zadnja - in najbolj zoprna - komplikacija: ker funkcija zelo zelo zelo velikokrat obiše isto točko (ne v okviru ene rekurzivne veje, temveč ker poskuša iste točke obiskovati v zelo veliko vrstnih redih), bo funkcija prepočasna. Dodati je potrebno `@cache`. To bi bilo trivialno, če si ne bi kot argument podajali množice `prepovedani` - `cache` shranjuje že videne argumente v slovar, zato se slabo razume s spremenljivimi argumenti. Množico zato uredimo in pretvorimo v terko - brez urejanja bi se nam zgodilo, da bi imeli dve množici, ki sta enaki, vendar imata različen vrstni red elementov (navidez, saj množice sploh ne poznajo koncepta vrstnega reda) in bi za dve množici, ki sta v bistvu enaki, dobili različne terke.

```
def dosegljive(x, y, d, n, zemljevid):
    @cache
    def doseg(x, y, obr, dovoljene):
        dos = set()
        if obr:
```

```

        for xn, yn in dovoljene:
            if 0 < abs(xn - x) + abs(yn - y) <= d:
                dos.add((xn, yn))
                dos |= dosege(xn, yn, obr - 1, tuple(sorted(set(dovoljene) - {(xn, yn)})))
    if obr < n:
        for xn, yn in izpostave:
            if 0 < abs(xn - x) + abs(yn - y) <= d:
                dos |= dosege(xn, yn, n, dovoljene)
    return dos

izpostave = zemljevid["*"]
tocke = set().union(*(v for k, v in zemljevid.items() if k != "*"))
return dosege(x, y, n, tuple(sorted(tocke)))

```

Obvoz okrog zadnje težave je, da argument `dovoljene` odstranimo. Pred rekurzivnim klicem iz `tocke` umaknemo trenutno koordinato in jo po klicu dodamo nazaj. To je v resnici tudi precej hitrejše, saj se izognemo vsem kopiranjem, pretvarjanjem in urejanjem.

```

def dosegljive(x, y, d, n, zemljevid):
    @cache
    def dosege(x, y, obr):
        dos = set()
        if obr:
            for xn, yn in tocke:
                if 0 < abs(xn - x) + abs(yn - y) <= d:
                    dos.add((xn, yn))
                    tocke.remove((xn, yn))
                    dos |= dosege(xn, yn, obr - 1)
                    tocke.add((xn, yn))
        if obr < n:
            for xn, yn in izpostave:
                if 0 < abs(xn - x) + abs(yn - y) <= d:
                    dos |= dosege(xn, yn, n)
        return dos

    izpostave = zemljevid["*"]
    tocke = set().union(*(v for k, v in zemljevid.items() if k != "*"))
    return dosege(x, y, n)

```

Rešitev z zemljevidom v obliki vrstic

Predpostavljam, da je večina študentov prebrala zemljevid kar v seznam vrstic. Pri sestavljanju naloge te različice, priznan, nisem poskusil. Izkaže se, da ni tako grozna.

6

```
def preberi_zemljevid(ime_dat):
    zemljevid = []
    for v, vrstica in enumerate(open(ime_dat)):
        zemljevid.append(vrstica.strip())
    return zemljevid

def najblizji(x, y, c, zemljevid, prepovedani):
    najkoord, najrazdalja = None, None
    for v, vrstica in enumerate(zemljevid):
        for s, znak in enumerate(vrstica):
            if znak == c or c == "" and znak != ".":
                razdalja = abs(s - x) + abs(v - y)
                if (s, v) != (x, y) and (s, v) not in prepovedani and (
                    najrazdalja is None
                    or razdalja < najrazdalja
                    or razdalja == najrazdalja and (s, v) < najkoord):
                    najkoord, najrazdalja = (s, v), razdalja
    return najkoord

def vsi_najpogostejsi(x, y, d, zemljevid):
    pojavitve = defaultdict(int)
    for v in range(max(0, y - d), min(len(zemljevid), y + d + 1)):
        for s in range(max(0, x - d + abs(y - v)),
                        min(len(zemljevid[0]), x + d + 1 - abs(y - v))):
            pojavitve[zemljevid[v][s]] += 1
    pojavitve["."] = 0
    maxpojav = max(pojavitve.values(), default=0)
    return {k for k, v in pojavitve.items() if v == maxpojav > 0}

def najpogostejsi(x, y, d, zemljevid):
    return min(vsi_najpogostejsi(x, y, d, zemljevid),
               default=None)
```

7

```
def angelca(x, y, znamenitosti, zemljevid):
    prepovedani = set()
    korakov = 0
    for znamenitost in znamenitosti:
        nasl = najblizji(x, y, znamenitost, zemljevid, prepovedani)
```



```

        if not nasl:
            break
        xn, yn = nasl
        korakov += abs(xn - x) + abs(yn - y)
        x, y = xn, yn
        prepovedani.add((x, y))
    return korakov

```

8

```

def johanca(x, y, pot, zemljevid):
    znamenitosti = ""
    obiskane = set()
    stevilka = ""
    for c in pot:
        if c in "<>^v":
            stev = int(stevilka or 1)
            x, y = {"<": (x - stev, y), ">": (x + stev, y), "v": (x, y + stev), "^": (x, y - stev)}[c]
            if (x, y) not in obiskane:
                obiskane.add((x, y))
                if 0 <= y < len(zemljevid) and 0 <= x < len(zemljevid[0]) and zemljevid[y][x] != ".":
                    znamenitosti += zemljevid[y][x]
            stevilka = ""
        else:
            stevilka += c
    return znamenitosti

```

9

```

def najboljsa_cetrta(a, zemljevid):
    maxn = 0
    for x in range(max(1, len(zemljevid[0]) - a + 1)):
        for y in range(max(1, len(zemljevid) - a + 1)):
            n = sum(len(vrstica[x:x+a].replace(".", "")) for vrstica in zemljevid[y:y+a])
            if n > maxn:
                maxn = n
                xx, yy = x, y
    return xx, yy

```

10

Te pa se mi ne da narediti na ta način. :)

Zanimivo je predvsem, da je Angelca - vsaj na mojem računalniku - dovolj hitra

tudi na ta način. Sicer je desetkrat počasnejša od one v gornjih rešitvah (pri meni potrebuje 1,2 s namesto 0,12 s), vendar je to vsaj pri meni še vedno v okviru omejitev.