

Day 17: Conway Cubes

(Povezava na nalogo)

V tej nalogi je bilo potrebno simulirati Conwayevo igro življenja v treh oziroma štirih dimenzijah. Ker smo frajerji, bomo napisali program, ki dela v (načelno) poljubnem številu dimenzij.

Pravil igre ne bomo razlagali. Preberite v opisu naloge ali na Wikipediji (gornje povezave).

Tule je celotna rešitev, za teaser. Po kosih je razložena spodaj.

```
from collections import defaultdict
from itertools import product

def neighbours_function(d):
    def neighs(coord):
        return {tuple(c + d for c, d in zip(coord, ds))
                for ds in dss}

    dss = list(product((-1, 0, 1), repeat=d))
    return neighs

def read_data(fname, d):
    adds = (0,) * (d - 2)
    cells = {(x, y) + adds
              for y, v in enumerate(open("input.txt")) for x, c in enumerate(v)
              if c == "#"}

    return cells

def next_gen(cells, neighs):
    counts = defaultdict(int)
    for cell in cells:
        for neigh in neighs(cell):
            counts[neigh] += 1

    return {coords for coords, neighs in counts.items()
            if neighs in ((3, 4) if coords in cells else (3,))}

for d in (3, 4):
    neighs = neighbours_function(d)
    cells = read_data("input.txt", d)
    for _ in range(6):
```

```

        cells = next_gen(cells, neighs)
    print(d, len(cells))
3 213
4 1624

```

Funkcija za izračun koordinat sosedov

Najprej bomo pripravili funkcijo, ki prejme terko s koordinatami (terka ima toliko elementov, kolikor dimenzij imamo) in vrne množico koordinat vseh sosed.

Najpreprostejši način vodi prek funkcije `itertools.product`, ki računa kartez-
ični produkt več množic (po domače: vse "kombinacije" stvari iz podanih sez-
namov). Lahko pa jih podamo eno samo reč in število ponovitev, pa vrne seznam
terk podane dolžine, kateri elementi prihajajo iz ... Uh, raje pokažem kar primer.

```

from itertools import product

list(product("abc", repeat=3))

[('a', 'a', 'a'),
 ('a', 'a', 'b'),
 ('a', 'a', 'c'),
 ('a', 'b', 'a'),
 ('a', 'b', 'b'),
 ('a', 'b', 'c'),
 ('a', 'c', 'a'),
 ('a', 'c', 'b'),
 ('a', 'c', 'c'),
 ('b', 'a', 'a'),
 ('b', 'a', 'b'),
 ('b', 'a', 'c'),
 ('b', 'b', 'a'),
 ('b', 'b', 'b'),
 ('b', 'b', 'c'),
 ('b', 'c', 'a'),
 ('b', 'c', 'b'),
 ('b', 'c', 'c'),
 ('c', 'a', 'a'),
 ('c', 'a', 'b'),
 ('c', 'a', 'c'),
 ('c', 'b', 'a'),
 ('c', 'b', 'b'),
 ('c', 'b', 'c'),
 ('c', 'c', 'a'),
 ('c', 'c', 'b'),
 ('c', 'c', 'c')]

```

Funkcijo bomo uporabili zato, da sestavi vse koordinate sosed - relativno glede na poljubno koordinato. V treh dimenzijah bo to videti takole:

```
dss = list(product((-1, 0, 1), repeat=3))
```

```
dss
```

```
[(-1, -1, -1),  
 (-1, -1, 0),  
 (-1, -1, 1),  
 (-1, 0, -1),  
 (-1, 0, 0),  
 (-1, 0, 1),  
 (-1, 1, -1),  
 (-1, 1, 0),  
 (-1, 1, 1),  
 (0, -1, -1),  
 (0, -1, 0),  
 (0, -1, 1),  
 (0, 0, -1),  
 (0, 0, 0),  
 (0, 0, 1),  
 (0, 1, -1),  
 (0, 1, 0),  
 (0, 1, 1),  
 (1, -1, -1),  
 (1, -1, 0),  
 (1, -1, 1),  
 (1, 0, -1),  
 (1, 0, 0),  
 (1, 0, 1),  
 (1, 1, -1),  
 (1, 1, 0),  
 (1, 1, 1)]
```

Funkcija, ki prejme neke koordinate in vrne koordinate vseh sosedov, je potem takšna:

```
def neighs(coord):  
    return {tuple(c + d for c, d in zip(coord, ds))  
            for ds in dss}
```

Odmike, kakršne smo sestavljali zgoraj, shranimo v **dss**. Funkcija **neighs** prejme terko s koordinatami in vrne seznam terk, katerih elementom so prišteti različni odmiki.

```
neighs((80, 5, 100))  
{(79, 4, 99),
```

```

(79, 4, 100),
(79, 4, 101),
(79, 5, 99),
(79, 5, 100),
(79, 5, 101),
(79, 6, 99),
(79, 6, 100),
(79, 6, 101),
(80, 4, 99),
(80, 4, 100),
(80, 4, 101),
(80, 5, 99),
(80, 5, 100),
(80, 5, 101),
(80, 6, 99),
(80, 6, 100),
(80, 6, 101),
(81, 4, 99),
(81, 4, 100),
(81, 4, 101),
(81, 5, 99),
(81, 5, 100),
(81, 5, 101),
(81, 6, 99),
(81, 6, 100),
(81, 6, 101)}

```

Vse, kar smo naredili doslej, zložimo v funkcijo `neighbours_function`, ki prejme število dimenzij in vrne funkcijo `neighs` za iskanje sosedov v takšnem številu dimenzij.

```

def neighbours_function(d):
    def neighs(coord):
        return {tuple(c + d for c, d in zip(coord, ds))
                for ds in dss}

    dss = list(product((-1, 0, 1), repeat=d))
    return neighs

```

Branje podatkov

Funkcija, ki prebere podatke, razbere koordinate iz datoteke, poleg tega pa doda še toliko ničel, kolikor je potrebno glede na podano število dimenzij.

```

def read_data(fname, d):
    adds = (0,) * (d - 2)
    cells = {(x, y) + adds

```

```

        for y, v in enumerate(open(fname))
        for x, c in enumerate(v)
        if c == "#"}

    return cells

```

Izračun naslednje generacije

Najprej bomo sestavili slovar, katerega ključi bodo koordinate, vrednosti pa število celic, ki živijo na tej celici (to je 1 ali 0) in vseh njenih sosedih.

To je preprosto: gremo čez vse celice, za vsako celico gremo čez vse njene sosede in jim prištejemo 1.

Tudi nadaljevanje je trivialno: v novi generaciji bodo celice na vseh koordinatah iz tega slovarja, pri katerih je število sosedov 3 ali 4, če je tam že prej bila celica (ker med sosede štejemo tudi celico samo!) oz. 3, če tam še ni bilo celice.

```

from collections import defaultdict

def next_gen(cells, neighs):
    counts = defaultdict(int)
    for cell in cells:
        for neigh in neighs(cell):
            counts[neigh] += 1

    return {coords for coords, neighs in counts.items()
            if neighs in ((3, 4) if coords in cells else (3,))}

```

Rešitev

Rešimo nalogo za 2, 3, 4 in 5 dimenzij.

Za 5 traja malo dlje, ker imamo v predzadnjem koraku več kot 22 tisoč celic, tako da je potrebno pregledati 5 milijonov sosednjih polj.

```

for d in (2, 3, 4, 5):
    neighs = neighbours_function(d)
    cells = read_data("input.txt", d)
    for _ in range(6):
        cells = next_gen(cells, neighs)
    print(d, len(cells))

```

```

2 35
3 213
4 1624
5 9516

```