resitev-checkpoint

January 28, 2024

1 Day 24: Lobby Layout

(Povezava na nalogo)

Imamo šestkotno mrežo. Možne smeri so sw, se, nw, ne, e in w. Da bi jih bilo bolj zabavno brati, so navodila za poti zbite skupaj. Recimo

[1]:
```
path = "seneewnwse"
```

je potrebno razbiti v `["se", "ne", "e", "w", "nw", "se"]`.

Takšne stvari se seveda da delati z regularnimi izrazi, vendar bomo naredili nekaj bolj zabavnega: generator, ki prejme takle niz in generira njegove dele.

[2]:
```
def split(path):
    path = iter(path.strip())
    for c in path:
        if c not in "ew":
            c += next(path)
        yield c
```

Od poti odluščimo morebitni beli prostor (morebitni `\n` na koncu). Nato ga spremenimo v iterator, da bomo lahko šli čezenj z zanko, poleg tega pa še z `next`. Za vsak znak preverimo, ali je e ali w. Če ni, dodamo še naslednji znak (`next(path)`).

Zdaj lahko preberemo navodila v generator generatorjev.

[3]:
```
instructions = map(split, open("input.txt"))
```

1.1 Prvi del

Dobimo seznam takšnih poti. Vsako prehodimo iz začetne pozicije. Polje, na katerega prispemo, spremenimo iz belega na črno ali iz črnega na belo (če je bilo poprej črno).

Vprašanje za prvi del je: koliko črnih polj imamo.

[4]:
```
from collections import defaultdict

directions = {"e": 1, "ne": 1j, "nw": -1 + 1j, "w": -1, "sw": -1j, "se": 1-1j}

flips = defaultdict(int)
```

```
for path in instructions:
    flips[sum(directions[step] for step in path)] += 1

black = {c for c, f in flips.items() if f % 2 == 1}

print(len(black))
```

```
538
```

V `directions` naberemo spremembe koordinat ob vsaki smeri. Če sta e in w 1 in -1, potem sta ne in nw i in $-1+i$, sw in se pa $-i$ in $1-1j$.

Prednost uporabe kompleksnih koordinat je, da jih je lahko seštevati. Z `directions[step] for step in steps` dobimo vse spremembe na poti `path` in s `sum` jih seštejemo. Tako dobimo končne koordinate. V `flips` štejemo, kolikokrat prispemo na posamezno polje. Črna so potem tista polja, ki smo jih obrnili lihokrat. Njihove koordinate spravimo v `black`, za kasnejšo rabo.

Rešitev prvega dela je velikost te množice.

1.2 Drugi del

V drugem delu igramo igro življenja na šestkotni mreži. Pravila so takšna: - črno polje ostane črno, če ima enega ali dva črna soseda in - belo polje postane črno, če ima natančno dva črna soseda.

Najprej si pripravimo funkcijo `neighbours(c)`, ki za polje z danimi koordinatami vrne množico koordinat sosednjih polj.

[5]:
```
n_dirs = tuple(directions.values())

def neighbours(c):
    return {c + x for x in n_dirs}
```

Odtod je preprosto napisati funkcijo, ki za dane koordinate vrne število črnih sosedov: to je pač velikost preseka množice koordinat črnih polj in koordinat sosedov.

[6]:
```
def count_neighbours(c):
    return len(neighbours(c) & black)
```

Zdaj pa sto generacij igre, kot zahteva naloga. V vsakem koraku izračunamo novo množico črnih polj. - Črna so vsa tista polja, ki so bila črna že prej in imajo enega ali dva črna soseda. - Poleg tega pa gremo čez vse sosede vseh črnih polj. Ta dobimo tako, da za vsak element množice `black` pokličemo `neighbours`, kar dobimo pa damo kot argument funkciji `set.union`. Od te množice odštejemo vsa polja, ki so že črna. Nato sestavimo množico vseh belih sosedov črnih polj, ki imajo vsaj dva črna soseda.

Unija teh dveh množic so nova črna polja.

[7]:
```
from itertools import chain

for _ in range(100):
    black = {c for c in black if count_neighbours(c) in (1, 2)} \
```

```
        | {c for c in set.union(*map(neighbours, black)) - black if␣
 ↪count_neighbours(c) == 2}

print(len(black))
```

```
4259
```