

## Day 08: Handheld Halting

(Povezava na nalogo)

Imamo procesor z enim samim registrom in tremi ukazi. `add op` poveča register za podano vrednost, `jmp op` je relativni skok za podano razdaljo, `nop op` ne naredi ničesar.

Podan je program v obliki

```
nop +0
acc +1
jmp +4
acc +3
jmp -3
acc -99
acc +1
jmp -4
acc +6
```

Program se zacikla. Prva naloga je ugotoviti vrednost registra ob prvem ponovljenem ukazu. Druga naloga je ugotoviti, kateri `jmp` je potrebno zamenjati v `nop` ali obratno, tako da se program izteče, to je, poskuša izvesti ukaz, ki je ontran programa. Naloga je, spet, poiskati vrednost registra v trenutku, ko se to zgodi.

### Branje podatkov

Najprej razkosamo vse vrstice na dvoje. To storimo z `map(str.split, open("input.txt"))`. `map`, kot morda vemo, ki prejme dva elementa, funkcijo in generator. Nato vsakemu elementu, ki ga vrne generator, pokliče funkcijo in vrača (kot generator) njene rezultate.

Kaj je `str.split`, pa vemo: nevezana metoda `split` razreda `str`. Če tega stavka ne razumemo, se lahko o tem poučimo ob rešitvi dneva 6.

Čez `map` bomo šli s `for` zanko: `for instr, op in map(str.split, open("input.txt"))`. Te `instr` in `arg` zlagamo v pare, ki pa jih - zaradi kasnejše rabe - ne bomo shranili v terko, temveč v seznam, pri čemer `op` spremenimo v `int`, torej `[instr, int(op)]`. Vse skupaj zložimo v seznam, ki predstavlja naš par.

```
program = [[instr, int(op)]
            for instr, op in map(str.split, open("input.txt"))]

program[:6]

[['jmp', 232],
 ['acc', 21],
 ['nop', 120],
 ['jmp', 239],
 ['acc', 18],
```

```
['acc', 41]]
```

Nič posebnega, je pa kratko in elegantno.

## Simulacija procesorja

Ta ni popolnoma nič posebnega. Vedno, ko simuliramo procesor (lani je bilo to potrebno početi v polovici nalog; procesor je bil vedno boljši in naloge vedno zanimivejše), potrebujemo *programski števec* (*program counter*) in *akumulator* (*accumulator*), pri nas bo sta to `pc` in `acc`. V tej konkretni nalogi pa potrebujemo seznam že obiskanih lokacij (`visited`), da bomo lahko zaznali prvo, ki se ponovi.

Program bo tekkel, dokler se lokacija še ni ponovila in je programski števec še znotraj programa.

V zanki pa imamo tri `if`-e, ki ustrezajo trem ukazom. Pri tem bodimo pozorni, da `jmp` spremeni `pc` in s `continue` preskoči ostanek, medtem ko se po preostalih ukazih izvede tudi `pc += 1`. Če bomo morali procesor simulirati še v kateri nalogi, bo tega še veliko, saj bomo najbrž dobili tudi pogojne skoke.

Funkcija vrne dve stvari: pove vrednost akumulatorja in ali se je program ustavil.

```
def execute(program):
    pc = 0
    acc = 0
    visited = set()
    while pc not in visited and pc < len(program):
        visited.add(pc)
        instr, arg = program[pc]
        if instr == "jmp":
            pc += arg
            continue
        if instr == "acc":
            acc += arg
        elif instr == "nop":
            pass
        pc += 1

    return acc, pc >= len(program)
```

Blok

```
elif instr == "nop":
    pass
```

očitno ne naredi ničesar in je nepotreben. Vendar mi je všeč, da ga napišemo, saj je tako očitno, da ta možnost obstaja in da je nismo pozabili.

## Prvi del

```
last_acc, _ = execute(program)
print(last_acc)

1528
```

## Drugi del

Drugi del tudi ni nič posebnega, razen tega, da se lahko izzivljamo s tem, kako bomo spremenili `nop` v `jmp` in obratno.

```
nop_jump = {"nop", "jmp"}
for mem in program:
    instr = mem[0]
    if instr in nop_jump:
        mem[0] = (nop_jump - {instr}).pop()
        acc, ok = execute(program)
        if ok:
            print(acc)
            break
    mem[0] = instr

640
```