resitev

January 28, 2024

1 Day 19: Monster Messages

(Povezava na nalogo)

Imamo zbirko pravil. - Nekatera pravijo, da je potrebno napisati črko; pravilo 1 tako pravi, naj napišemo a. - Nekatere pravijo, da moramo uporabiti zaporedje pravil; pravilo 0 veleva uporabiti pravilo 1 in nato pravilo 2. - Nekatera pa nam puščajo izbiro. Pravilo 2 pravi, da lahko uporabimo 1 in 3 ali pa 3 in 1.

```
0: 1 2
1: "a"
2: 1 3 | 3 1
3: "b"
```

Takšni stvari se reče kontekstno neodvisna gramatika. S takšnimi gramatikami so definirani programski jeziki. Program v nekem jeziku je sintaktično pravilen, če ustreza pravilom gramatike tega jezika. V nalogi bo potrebno preverjati, katere besede ustrezajo podani gramatiki.

Gornja gramatika opisuje nize `aab` in `aba`. V vsakem primeru začnemo z 1, ki izpiše a, nato pa nadaljujemo z 1 3 ali 3 1, ki izpišeta a in b ali b in a.

1.1 Branje podatkov

V datoteki je seznam pravil in seznam sporočil, kateri pravilnost bo potrebno preverjati, na primer

```
0: 4 1 5
1: 2 3 | 3 2
2: 4 4 | 5 5
3: 4 5 | 5 4
4: "a"
5: "b"

ababbb
bababa
abbbab
aaabbb
aaaabbb
```

[1]:
```
rules_part, messages_part = open("example.txt").read().split("\n\n")

rules = {}
```

```
for line in rules_part.splitlines():
    i, subrules = line.split(": ")
    if subrules[0] == '"':
        subrules = subrules[1]
    else:
        subrules = [[int(x) for x in part.split()]
                    for part in subrules.split("|")]
    rules[int(i)] = subrules

messages = [x.strip() for x in messages_part.splitlines()]
```

Kot v že nekaj nalogah tudi tu preberemo datoteko in jo razdelimo glede na `"\n\n"`.

Del s pravili razbijemo na vrstice, vsako vrstico razdelimo glede na `": "`. Če se del desno od dvopičja začne z narekovajem, si zapomnimo prvi znak. Sicer pa ga razbijemo glede na `"|"`, vsak del pa nato glede na beli prostor in na koncu pretvorimo v `int`.

(Kot vidite smo tole branje podrobneje opazovali v prvih nalogah. Tisti, ki se prebijajo čez te naloge, najbrž že znajo dovolj, da jih tole ne vznemirja preveč.)

[2]:
```
rules
```

[2]:
```
{0: [[4, 1, 5]],
 1: [[2, 3], [3, 2]],
 2: [[4, 4], [5, 5]],
 3: [[4, 5], [5, 4]],
 4: 'a',
 5: 'b'}
```

Rezultat je torej slovar. Ključi so številke pravil, vrednosti pa bodisi znak (izkaže se, da imamo samo a in b, vendar bi bilo funkcije možno brez težav posplošiti na kaj daljšega), bodisi seznam seznamov, ki predstavljajo možna zaporedja pravil. (Če kdo ne razume, naj primerja ta slovar s podatki v datoteki in vse bo jasno.)

Sporočila le razbijemo na vrstice in odstranimo beli prostor na koncih.

2 Prvi del: končna gramatika

V pravilih ni ciklov. To zelo zelo zelo poenostavi naše delo. Nizi so končnih dolžin in jih je, posledično, končno mnogo. (Če bi bila dolžina (in abeceda) omejena, bi zelo težko sestavili neskončno različnih besedil.)

Naloga je preveriti, katera od sporočil ustrezajo prvim pravilom (in, seveda, vsem, na katera se prvo pravilo sklicuje).

Tule je funkcija za preverjanje pravilnosti začetka sporočila.

[3]:
```
def check(rule, s):
    if isinstance(rule, str):
        if s[0] == rule:
```

```
            return s[1:]
        else:
            return None

    for option in rule:
        remaining = s
        for subrule_no in option:
            remaining = check(rules[subrule_no], remaining)
            if remaining is None:
                break
        else:
            return remaining
    return None
```

- Če sporočilo ne ustreza pravilu, funkcija vrne `None`.
- Če začetek sporočila ustreza pravilu, pa funkcija vrne neporabljeni del niza. Če bi, recimo, preverjali, ali niz `"babba"` ustreza pravilu 3, bi funkcija vrnila `"bba"`. Prvi dve črki sta namreč porabljeni, `"bba"` pa ostane.

[4]:
```
check(rules[3], "babba")
```

[4]:
```
'bba'
```

Tule predpostavljamo, da lahko (začetek) sporočila ustreza pravilom le na en način. Izkaže se, da pri tej nalogi to deluje. Kako je s tem v splošnem, bi vam z veseljem napisal, če ne bi tega že davno pozabil.

V prvem delu poskrbimo za pravila, ki vsebujejo le en znak. Če je torej pravilo en sam znak, preverimo, ali je enak prvemu znaku niza. Če je, vrnemo ostanek niza. Če ni, vrnemo `None`, ker začetek niza očitno ne ustreza pravilu.

V drugem delu gremo prek različnih opcij. Vsaka je sestavljena iz zaporedja pravil. Gremo čez vsa ta podpravila in za vsako preverimo, ali ustreza začetku - s tem, da vse, kar posamezno pravilo porabi, sproti odbijamo.

```
remaining = s
for part in option:
    remaining = check(rules[part], remaining)
```

Če katero od podpravilo javi, da sporočilo ne ustreza, z `break` prekinemo zanko prek zaporedja pravil in nadaljujemo z naslednjo opcijo. Če se zanka prek podpravil izteče, pa vrnemo preostanek niza. (Tu naredimo predpostavko, da je to edino možno zaporedje podpravil. Če ne bi bilo tako, bi se program nekoliko zapletel, vendar bi preživeli: funkcijo bi napisali tako, da vrne vse možne ostanke, potem pa naj se ta, ki jo kliče, znajde, kakor hoče.)

Končno, če se zanka pred opcij izteče, ne da bi našli katero, ki je pravilna, vrnemo `None`.

Naloga hoče, da preštejemo sporočila, ki ustrezajo pravilu 0. Se pravi vsa sporočila, za katerega `check(rule0, message)` vrne prazen niz.

[5]:
```
rule0 = rules[0]
print(sum(check(rule0, message) == "" for message in messages))
```

```
2
```

Zdaj pa nadaljujmo s pravimi podatki.

[6]:
```
rules_part, messages_part = open("input.txt").read().split("\n\n")

rules = {}
for line in rules_part.splitlines():
    i, subrules = line.split(": ")
    if subrules[0] == '"':
        subrules = subrules[1]
    else:
        subrules = [[int(x) for x in part.split()]
                    for part in subrules.split("|")]
    rules[int(i)] = subrules

messages = [x.strip() for x in messages_part.splitlines()]
```

[7]:
```
rule0 = rules[0]
print(sum(check(rule0, message) == "" for message in messages))
```

```
104
```

2.1 Intermezzo: Vsa možna sporočila

Bi znali sestaviti vsa možna sporočila? Znali.

[8]:
```
from itertools import product, count

def all_messages(rule):
    if isinstance(rule, str):
        return [rule]

    messages = []
    for option in rule:
        submessages = [all_messages(rules[part]) for part in option]
        messages += ["".join(comb) for comb in product(*submessages)]
    return messages
```

Funkcija prejme pravilo. Če je pravilo zgolj črka, je seznam vseh možnih pravil le seznam, ki vsebuje to črko.

Sicer pa gremo čez vse opcije. Za vsako gremo čez vsa podpravila, z `all_messages(rules[part])` pogledamo, kaj to pravilo zgenerira. Če imamo `0: 1 4 5`. Bo `submessages` seznam, ki bo vseboval tri sezname: v prvem bo vse, kar sestavi pravilo 1, v drugem vse, kar sestavi pravilo 4 in v tretjem vse, kar sestavi pravilo 5. Če lahko pravilo 1 sestavi le niza `"ba"` in `"aba"`, pravilo 4 niza `"b"` in

`"abb"`, pravilo 5 pa samo `"a"`, bomo imeli `[["ba", "aba"], [""b", "abb"], ["a"]]`. Iz tega je potrebno sestaviti vse kombinacije "ba-b-a", "ba-abb-a", "aba-abb-a", "aba-abb-a" - seveda brez vezajev, ti so tu samo, da lažje razumemo, kaj počnemo. Natančno to nam naredil funkcija `product`. Gremo torej čez vse te kombinacije in jih združimo z `join`.

Zdaj znamo prvi del rešiti še enkrat: zanima nas velikost presek podanih sporočil in sporočil, ki jih je mogoče sestaviti iz pravila 0.

[9]:
```
print(len(set(messages) & set(all_messages(rules[0]))))
```

```
104
```

2.2 Drugi del: neskončna gramatika

Tu postanejo stvari nekoliko zabavnejše, a ne preveč: pravili 8 in 11 je potrebno spremeniti tako:

```
8: 42 | 42 8
11: 42 31 | 42 11 31
```

Vprašanje pa ostaja isto: koliko pravil lahko sestavimo iz pravila 0?

Pravilo `0` se, kakšno naključje, glasi

```
0: 8 11
```

Pravili 8 in 11 se, kakšna sreča, ne pojavljata v nobenem drugem pravilu. (Sicer bi stvari ne bile samo nekoliko zabavnejše, temveč preveč zabavnejše.)

Pravilo 8 v bistvu pravi, da smemo poljubnokrat (a vsaj enkrat!) ponoviti pravilo 42. Dovoljena zaporedja so, recimo `42`, `42 42`, `42 42 42`, `42 42 42 42`…

Pravilo 11 pravi, da smemo poljubnokrat ponoviti `42`, slediti pa mora enako število ponovitev `31`, recimo `42 31`, ali `42 42 31 31` ali `42 42 42 42 31 31 31 31`.

Ker `0` pravi, "najprej 8, potem 11", to pomeni, da bomo imeli najprej vsaj eno ponovitev `42`, nato pa enako število `42` in `31`.

Z drugimi besedami: pravilo 0 zahteva, da najprej ponavljamo 42, nato 31, pri čemer se mora 42 pojaviti vsaj dvakrat, 31 pa vsaj enkrat, vendar manjkrat, kot se je pojavila 42.

Zdaj pa moramo izvedeti, kaj ustvarita `31` in `42`. To nam pove funkcija iz intermezza.

[10]:
```
r42 = set(all_messages(rules[42]))
r31 = set(all_messages(rules[31]))
```

[11]:
```
r42
```

[11]:
```
{'aaaaaaaa',
 'aaaaaaab',
 'aaaaabba',
 'aaaaabbb',
 'aaaabaaa',
 'aaaababa',
 'aaaababb',
```

```
'aaaabbab',
'aaaabbba',
'aaabaaab',
'aaabaaba',
'aaababaa',
'aaababab',
'aaabbbaa',
'aabaaaba',
'aabaaabb',
'aabaabaa',
'aabaabba',
'aabaabbb',
'aababaab',
'aabababa',
'aababbaa',
'aabbaaaa',
'aabbaaba',
'aabbaabb',
'aabbabab',
'aabbbaaa',
'aabbbaab',
'aabbbaba',
'aabbbabb',
'aabbbbbb',
'abaaaaab',
'abaaaabb',
'abaaabaa',
'abaaabba',
'abaaabbb',
'abaabaab',
'abaabbba',
'abaabbbb',
'ababaaba',
'abababab',
'ababABba',
'abABbaaa',
'abABbaab',
'ababbaba',
'abABbabb',
'abABbbab',
'abABbbba',
'abABbbbb',
'abbabaaa',
'abbabaab',
'abbabbba',
'abbabbbb',
'abbbaaba',
```

```
'abbbabaa',
'abbbabba',
'abbbabbb',
'abbbbaab',
'abbbbaba',
'abbbbabb',
'abbbbbab',
'abbbbbbb',
'baaaaaaa',
'baaaaaab',
'baaaaabb',
'baaabaaa',
'baaababa',
'baaababb',
'baaabbab',
'baaabbba',
'baaabbbb',
'baabaaaa',
'baabaaab',
'baabaabb',
'baababaa',
'baababba',
'baababbb',
'baabbaaa',
'baabbabb',
'baabbbaa',
'baabbbab',
'baabbbbb',
'babaaaaa',
'babaaaba',
'babaabaa',
'babaabba',
'babaabbb',
'bababaaa',
'bababaab',
'babababa',
'babababb',
'bababbaa',
'babbaabb',
'babbabaa',
'babbabab',
'babbbaaa',
'babbbaab',
'bbaaaaab',
'bbaaaabb',
'bbaaabab',
'bbaaabba',
```

```
 'bbaaabbb',
 'bbaabaaa',
 'bbaabbab',
 'bbaabbbb',
 'bbabaaaa',
 'bbabaaba',
 'bbababaa',
 'bbababab',
 'bbababba',
 'bbabbaaa',
 'bbabbaab',
 'bbabbaba',
 'bbabbbaa',
 'bbabbbab',
 'bbabbbba',
 'bbbaaaab',
 'bbbaabaa',
 'bbbabaab',
 'bbbababb',
 'bbbabbaa',
 'bbbbaaab',
 'bbbbaaba',
 'bbbbabaa',
 'bbbbbaab',
 'bbbbbaba',
 'bbbbbabb',
 'bbbbbbbb'}
```

[12]:
```
r31
```

[12]:
```
{'aaaaaaba',
 'aaaaaabb',
 'aaaaabaa',
 'aaaaabab',
 'aaaabaab',
 'aaaabbaa',
 'aaaabbbb',
 'aaabaaaa',
 'aaabaabb',
 'aaababba',
 'aaababbb',
 'aaabbaaa',
 'aaabbaab',
 'aaabbaba',
 'aaabbabb',
 'aaabbbab',
 'aaabbbba',
```

```
'aaabbbbb',
'aabaaaaa',
'aabaaaab',
'aabaabab',
'aababaaa',
'aabababb',
'aababbab',
'aababbba',
'aababbbb',
'aabbaaab',
'aabbabaa',
'aabbabba',
'aabbabbb',
'aabbbbaa',
'aabbbbab',
'aabbbbba',
'abaaaaaa',
'abaaaaba',
'abaaabab',
'abaabaaa',
'abaababa',
'abaababb',
'abaabbaa',
'abaabbab',
'ababaaaa',
'ababaaab',
'ababaabb',
'abababaa',
'abababbb',
'ababbbaa',
'abbaaaaa',
'abbaaaab',
'abbaaaba',
'abbaaabb',
'abbaabaa',
'abbaabab',
'abbaabba',
'abbaabbb',
'abbababa',
'abbababb',
'abbabbaa',
'abbabbab',
'abbbaaaa',
'abbbaaab',
'abbbaabb',
'abbbabab',
'abbbbaaa',
```

```
'abbbbbaa',
'abbbbbba',
'baaaaaba',
'baaaaabb',
'baaaabaa',
'baaaabab',
'baaaabba',
'baaabaab',
'baaabbaa',
'baabaaba',
'baababab',
'baabbaab',
'baabbaba',
'baabbbba',
'babaaaab',
'babaaabb',
'babaabab',
'bababbab',
'bababbba',
'bababbbb',
'babbaaaa',
'babbaaab',
'babbaaba',
'babbabba',
'babbabbb',
'babbbaba',
'babbbabb',
'babbbbaa',
'babbbbab',
'babbbbba',
'babbbbbb',
'bbaaaaaa',
'bbaaaaba',
'bbaaabaa',
'bbaaabab',
'bbaababa',
'bbaababb',
'bbaabbaa',
'bbaabbba',
'bbabaaab',
'bbabaabb',
'bbababbb',
'bbabbabb',
'bbabbbbb',
'bbbaaaaa',
'bbbaaaba',
'bbbaaabb',
```

```
 'bbbaabab',
 'bbbaabba',
 'bbbaabbb',
 'bbbabaaa',
 'bbbababa',
 'bbbabbab',
 'bbbabbba',
 'bbbabbbb',
 'bbbbaaaa',
 'bbbbaabb',
 'bbbbabab',
 'bbbbabba',
 'bbbbabbb',
 'bbbbbaaa',
 'bbbbbbaa',
 'bbbbbbab',
 'bbbbbbba'}
```

Pač neka množica nizov. Ti imajo dve lepi lastnosti.

Prva: vsi imajo 8 znakov.

[13]:
```
{len(x) for x in r31}
```

[13]:
```
{8}
```

[14]:
```
{len(x) for x in r42}
```

[14]:
```
{8}
```

Druga: nobeno zaporedje se ne pojavi v obeh.

[15]:
```
r31 & r42
```

[15]:
```
set()
```

Torej nas ne čaka nič težkega: jemljemo po osem znakov niza, gledamo, koliko osmeric se pojavi v `r42` in potem, koliko se jih v `r32`. Če se njihovo število ujema s tistim, kar smo napisali zgoraj, je sporočilo v redu. Če ne, ni.

Funkcija `count_block(message, msgset)` prejme sporočilo in množico dovoljenih blokov dolžine 8 in kot rezultat vrne število ponovitev teh blokov na začetku niza.

[16]:
```
from itertools import count

def count_block(message, msgset):
    for i in count():
        if message[8 * i:8 * (i + 1)] not in msgset:
            return i
```

Funkcija `check_message_42_31` prešteje, kolikokrat se pojavi kak blok iz `r42` in kolikokrat v nadaljevanju kak blok iz `r32`. Obojih skupaj mora biti toliko, da porabita ravno vse sporočilo (sicer so v sporočilu bloki, ki ju ni ne v enem ne v drugem), pa še število blokov mora ustrezati.

[17]:
```
def check_42_31(message):
    c42 = count_block(message, r42)
    c31 = count_block(message[c42 * 8:], r31)
    return c42 + c31 == len(message) // 8 and c42 > c31 and c42 >= 2 and c31 >=␣
 ↪1
```

[18]:
```
print(sum(map(check_42_31, messages)))
```

```
314
```