

Kako oddati?

Če boste rešili obvezno in dodatno nalogo, oddajte datoteko, v kateri je najprej dodatna naloga in šele nato obvezna. Boste videli, zakaj.

Neobvezni del je tokrat kar zafrknjen.

Obvezna naloga

Imamo štiri vrstice besedila. Shranjene so v spodnji tabeli, kje se začnejo, pa pove zgornja tabela.

Prva beseda se začne s tretjim elementom spodnje tabele, druga z dvanajstim, tretja z enajstim in četrta z osmim. Za primer, preberimo drugo vrstico.

Začne se, kot smo rekli, na **12.** mestu.

- Na 12. mestu izvemo, da je prva črka **B**, naslednjo pa nam je iskati na **14.** mestu.
- Na 14. mestu piše, da je to črka **e**, naslednjo pa izvemo na **13.** mestu.
- Na 13. mestu piše, da je to črka **r**, naslednjo pa nam je iskati na **7.** mestu.
- Na 7. mestu piše, da je to črka **t**, naslednjo pa nam je iskati na **6.** mestu.
- Na 6. mestu piše, da je to črka **a**, naslednjo pa nam je iskati na ... no, **-1** pomeni, da je to konec vrstice.

Na začetek programa napišite tabeli v takšni obliki:

```
zacetki = [3, 12, 11, 8]
potek = [(-1, 'a'), (15, 'n'), (10, 'a'), (1, 'A'), (9, 'l'), (4, 'i'), (-1, 'a'), (6, 't'),
          (2, 'D'), (0, 'k'), (16, 'n'), (5, 'C'), (14, 'B'), (7, 'r'), (13, 'e'), (-1, 'a')]
```

Program mora sestaviti seznam `besedilo`, ki vsebuje besedila vseh vrstic. Na konec programa dodajte `print(besedilo)`, da bo ta, ki popravlja, videl, da res deluje. Za gornji primer bi torej moral izpisati

```
['Ana', 'Berta', 'Cilka', 'Dani']
```

Ko to deluje, pa zamenjaj tabeli z naslednjima:

```
zacetki = [119, 178, 321, 180, 17]
potek = [
    (100, 'h'), (37, 'e'), (134, 'm'), (280, 'r'), (146, 'v'), (83, 'i'), (149, ' '), (80, ' '),
    (231, 'M'), (194, 'r'), (60, 's'), (40, 'o'), (237, 'k'), (6, 'i'), (92, 'a'), (170, 'e'),
    (58, 'i'), (11, 'k'), (44, 'm'), (171, 'r'), (126, 'u'), (151, 'd'), (248, 'i'), (98, 'v'),
    (168, 't'), (107, 'j'), (233, 'j'), (221, ' '), (65, 'a'), (257, 't'), (282, 'l'), (15, ' '),
    (184, 'v'), (84, ' '), (263, ' '), (117, 's'), (175, ' '), (19, ' '), (270, 'l'), (220, ' '),
    (46, 't'), (239, 'e'), (228, ' '), (71, 'v'), (268, 'a'), (104, 'i'), (271, ' '), (208, ' '),
    (308, 'a'), (72, 'e'), (223, ' '), (256, 'č'), (181, 'p'), (55, ' '), (261, 'o'), (47, ' '),
    (227, 'e'), (211, 'e'), (94, 'l'), (267, 'r'), (292, 't'), (190, 'r'), (251, 'g'), (197, ' '),
    (219, ' '), (309, ' '), (204, 'j'), (230, 'o'), (296, 'n'), (198, ' '), (5, 'n'), (77, ' '),
    (105, 'p'), (173, 'G'), (153, 'a'), (166, 'a'), (177, 'l'), (125, 'm'), (289, 'o'), (272, ' ')
]
```

```

(298, ' '), (185, 'v'), (209, 's'), (293, 'c'), (150, 'z'), (20, 'j'), (141, 'd'), (235,
(69, 'i'), (135, ' '), (301, 'č'), (130, ' '), (222, 'č'), (30, 'p'), (250, 'a'), (45,
(201, 'o'), (320, 'a'), (312, ' '), (96, 'r'), (258, 'l'), (137, ' '), (87, 'o'), (290,
(28, 'l'), (214, 'r'), (13, 'm'), (191, 'i'), (286, 'm'), (64, 'o'), (0, ' '), (278, 'j'),
(93, ' '), (43, 'd'), (262, 'p'), (139, ' '), (73, '.'), (303, 'a'), (240, 'h'), (102,
(158, ' '), (152, 'a'), (12, 'r'), (329, 'r'), (302, 'm'), (332, 'a'), (297, 'b'), (159,
(8, ' '), (226, 'e'), (129, 'j'), (313, 'r'), (70, 'č'), (327, ' '), (205, 'o'), (276,
(38, 'e'), (316, 'b'), (75, 'p'), (167, 'r'), (306, 'j'), (283, 'r'), (265, 's'), (236,
(122, 'a'), (50, 'k'), (22, 's'), (91, 'e'), (118, ' '), (97, 'z'), (113, ' '), (63, 'r'),
(323, 'v'), (144, 'd'), (109, 't'), (229, 't'), (311, 's'), (111, ' '), (32, 's'), (224,
(34, 'o'), (101, 'e'), (7, 'l'), (25, 'n'), (241, 'e'), (172, 'u'), (288, 'd'), (51, 'e'),
(112, 'o'), (279, 't'), (42, 't'), (90, 'e'), (163, 'k'), (121, 'l'), (154, 's'), (199,
(244, 'i'), (295, 'o'), (285, 'N'), (246, 'i'), (202, 'k'), (49, 'r'), (200, 'm'), (215,
(299, 'e'), (232, 'e'), (164, 'r'), (275, 'r'), (41, 'j'), (67, 'r'), (326, 'o'), (183,
(243, 'l'), (131, 'p'), (252, 'o'), (318, 'j'), (48, 'k'), (213, 'n'), (-1, ' '), (56,
(179, 'b'), (24, 's'), (59, 'e'), (147, 's'), (253, 'a'), (140, 'ž'), (305, 'r'), (242,
(68, 'r'), (266, 'o'), (143, ' '), (9, 'p'), (145, 'a'), (212, 'j'), (315, 'o'), (161,
(106, 'i'), (99, 'p'), (176, 'k'), (31, 's'), (325, ' '), (186, 'p'), (33, 'e'), (85,
(138, 'd'), (281, 'a'), (269, ' '), (189, 'p'), (322, 'z'), (210, 'a'), (287, 's'), (21,
(76, 't'), (89, 'e'), (82, 'e'), (4, ' '), (249, 'n'), (162, 'o'), (182, 'u'), (95, ' '),
(14, 'l'), (3, 'p'), (120, 'o'), (195, ' '), (2, ' '), (81, 's'), (114, ' '), (23, 'o'),
(273, ' '), (203, ' '), (-1, ' '), (115, 'a'), (169, 's'), (307, 'l'), (193, ' '), (234,
(133, ' '), (108, 'i'), (132, 'a'), (124, ' '), (319, 'e'), (196, 'n'), (57, 'r'), (238,
(324, 'a'), (192, 'i'), (128, 'r'), (317, ' '), (-1, '.'), (206, 'p'), (218, 'i'), (35,
(304, 'a'), (136, 'v'), (328, 'j'), (174, 'o'), (103, 'b'), (-1, '.'), (254, 'e'), (127,
(156, 'o'), (-1, '.'), (79, 'e'), (62, 'u'), (61, 'p'), (142, 'o'), (110, 'a'), (1, 't'),
(331, 'l'), (26, ' '), (66, 'i'), (245, ' '), (216, 'n'), (18, 'a'), (36, 'l'), (225, 'l'),
(10, 'o'), (294, 'i'), (78, 'j'), (16, 't'), (188, ' '), (277, 'i'), (300, 'u'), (264,
(155, 's'), (284, 'e'), (157, 'e'), (291, 'a'), (88, 'm'), (123, 'p'), (148, 'e'), (310,
(165, 's'), (255, 'o'), (260, 'r'), (29, 's'), (39, 'i'), (160, 's'), (27, 'e'), (187,
(54, 'p'), (274, 'N'), (53, ' '), (259, 'a'), (86, ' '), (314, 'p'), (207, 's'), (74, 'l'),
(330, 'e'), (217, 'e'), (247, 'g'), (116, 'i'), (52, ' ')

```

Poženi, preveri, oddaj - če rešuješ samo obvezno nalogo. Če tudi dodatno, pa beri naprej.

Rešitev

Pripravimo prazen seznam **besedilo**, v katerega bomo zlagali vrstice. Nato je potrebno iti čez tabelo **zacetki**, v kateri bomo dobivali začetne pozicije. Za vsako bomo sestavili celo vrstico in jo dodali v **besedilo**. Program bo torej v osnovi takšen:

```

besedilo = []
for pozicija in zacetki:
    vrstica = ""
    # tule moramo zdaj še dodati v vrstico vse znake, ki sodijo vanjo

```

```

...
besedilo.append(vrstica)

print(besedilo)

['', '', '', '', '']

```

Čez seznam **zacetki** smo spustili zanko **for**, saj bi radi nekaj naredili z vsakim elementom tega seznama. Znotraj te zanke bo očitno še ena zanka, tista, ki bo dodajala znake v **vrstica**. V tej zanki ne bomo šli čez seznam **potek**, kot je marsikdo (napačno) pomislil. Tu ne počnemo ničesar z *vsakim elementom seznama potek*, temveč skačemo po njem, dokler ne pridemo do indeksa **-1**. *Dokler* diši po zanki **while**, in to upravičeno.

Početi nam je torej tole: dokler pozicija ni enaka **-1**, preberemo par, ki se v **potek** nahaja na indeksu **pozicija**. Ta vsebuje naslednjo pozicijo in črko, ki jo je potrebno dodati. Pa imamo celotno rešitev:

```

besedilo = []
for pozicija in zacetki:
    vrstica = ""
    while pozicija != -1:
        pozicija, crka = potek[pozicija]
        vrstica += crka
    besedilo.append(vrstica)

print(besedilo)

```

```
['Kot vsi veliki možje je profesor Modrinjak ljubil preproste reči.', 'Nosil je preproste h
```

Pogoste napake in nepotrebni zapleti

while True Pogosto sem videval tole:

```

while True:
    ...
    if pozicija == -1:
        break

```

Zanka **while True** ni nič ilegalnega, pogosto jo v resnici potrebujemo, kadar se mora nekaj zgoditi vsaj enkrat (nekateri jeziki imajo zato zanko **do-while** ali **repeat-until**) ali kadar je izstop iz zanke nekoliko bolj zapleten. V programih, ki jih pišejo bolj začetniki, pa zanka **while True**, ki ima na koncu pogoj in **break**, tipično pomeni le, da avtor programa ni dovolj dobro razmislil, kaj hoče početi s to zanko. (Ali pa mu je nalogo reševal kak star C-jevec, ki bi tule delal (tudi v dobrih programih v C-ju ne prav običajno) zanko **do-while**.)

Reset na koncu Namesto

```

for pozicija in zacetki:
    vrstica = ""
    while pozicija != -1:
        pozicija, crka = potek[pozicija]
        vrstica += crka
    besedilo.append(vrstica)

```

sem velikokrat videl

```

vrstica = ""
for pozicija in zacetki:
    while pozicija != -1:
        pozicija, crka = potek[pozicija]
        vrstica += crka
    vrstica = ""
    besedilo.append(vrstica)

```

Za to sta spet možna dva razloga. Bodisi programer-začetnik na začetku programa nastavi začetne vrednosti spremenljivk in ko jih je potrebno pobrisati, jih pač pobriše, kjer je treba. Druga, manj všečna razlaga, je da gre za vzorec, ki bi ga uporabil slab programer v C++ ali podobnem jeziku (spremenljivko bi deklariral v zunanjem bloku, zato bi imela v začetku notranjega privzeto vrednost itn.)

Prvo je očitno boljše, lažje berljivo, bližje pravi semantiki programa.

Nerazpakirani pari V gornji rešitvi smo par iz `potek` razpakirali v dve spremenljivki in jima dali smiselni imeni `pozicija` in `crka`, se pravi, `pozicija, crka = potek[pozicija]`, tako da smo v nadaljevanju vedeli, s čim delamo.

Če tega ne storimo, je program videti tako:

```

while potek[zacetek][0] != -1:
    vrstica += potek[zacetek][1]
    zacetek = potek[zacetek][0]
    if potek[zacetek][0] == -1:
        vrstica += potek[zacetek][1]
    # in tako naprej

```

To je nepregledno, dolgo, težko berljivo. Kaj počne `vrstica += potek[zacetek][1]`? Ni boljše `vrstica += crka`? Spremenljivke imajo imena zato, da dokumentirajo program.

Vmesna različica je razpakiranje v spremenljivke s slabimi imeni in/ali razpakiranje z nepotrebnim indeksom. Z drugimi besedami, namesto

```
pozicija, crka = potek[pozicija]
```

pišemo

```
y = potek[x][1]
x = potek[x][0]
```

Vse skupaj Oboje skupaj potem izgleda tako, da imamo namesto

```
for pozicija in zacetki:
    vrstica = ""
    while pozicija != -1:
        pozicija, crka = potek[pozicija]
        vrstica += crka
    besedilo.append(vrstica)
```

tole:

```
ime = ""
for x in zacetki:
    while True:
        y = potek[x][1]
        x = potek[x][0]
        ime += y
        if x == -1:
            imena.append(ime)
            ime = ""
            break
```

Najboljše, da kar ponovimo, zakaj vse je to slabo:

- `ime = ""` daje vtis, da je `ime` spremenljivka, ki se nekako tiče celotnega programa, v resnici pa se `ime` rodi, uporabi in umre znotraj vsakega kroga zanke. (Poleg tega je `ime` slabo ime spremenljivke. Gre za besedo ali vrstico, v splošnem.)
- `for x in zacetki:` `x` je slabo, neinformativno ime spremenljivke.
- `while True` ne nudi informacije o tem, koliko časa se bo ta zanka vrtela. `while x != -1` bi povedal, da dokler bo `x` različen od `-1`.
- `y = potek[x][1]`: tudi `y` je slabo, neinformativno ime spremenljivke. Poleg tega tak način razpakiranja ni lep. `pozicija, crka = potek[pozicija]` pove, da element `potek[pozicija]` vsebuje dve stvari (novo pozicijo in črko). Če razpakiramo z indeksi, pa ob branju ni takoj očitno, da `potek[pozicija]` vsebuje natančno dve stvari; lahko bi jih tudi več. Z drugimi besedami, `pozicija, crka = potek[pozicija]` nam pove veliko, `y = potek[x][1]`; `x = potek[x][0]` nam ne pove skoraj nič.
- `if x == -1`: to bi, kot rečeno, moral biti pogoj v `while`.
- `imena.append(ime)`: tole je še posebej zanimiva pomanjkljivost. Sugerira, da je `imena.append(ime)` nekaj, kar se dogaja znotraj zanke (enako `ime`

= ""). Če vidimo, da jima sledi `break` in malo razmislimo, spoznamo, da se to zgodi na koncu zanke, oziroma, v bistvu, *po njej*. Tako je tudi smiselno: `ime` dodamo v `imena` takrat, ko je le-to sestavljeno. Najmanj, kar bi tu morali popraviti, je vsaj tole:

```
ime = ""
for x in zacetki:
    while True:
        y = potek[x][1]
        x = potek[x][0]
        ime += y
        if x == -1:
            break
    imena.append(ime)
ime = ""
```

Odvečne spremenljivke Tole je bilo zelo tipično:

```
for zacetek in zacetki:
    vrstica = ''
    pozicija = zacetek

    while pozicija != -1:
        nova_pozicija, crka = potek[izhodisce]
        ime += b
        pozicija = nova_pozicija
    besedilo.append(vrstica)
```

Tu imamo tri spremenljivke (`zacetek`, `pozicija` in `nova_pozicija`), čeprav zadošča le ena.

Čemu ločevati `zacetek` in `pozicija`? Slutim, da kakega študenta malo skrbi, da se bo zanka `for` "izgubila", če bomo pokvarili vrednost `zacetek`. (Tak strah bi gotovo ne bil nepričakovan za nekoga, ki zanko `for` pozna iz C-ja in podobnih jezikov.) Strah je odveč: zanka `for` bo mirno priredila spremenljivki `zacetek` novo vrednost iz seznama, ne glede na to, kaj počnemo z njo.

Čemu pa ločevati `nova_pozicija` in `pozicija`? Spet lahko le ugibam. Morda gre za strah; študentu je neprijetno kar tako, mimogrede, spremeniti vrednost `pozicija`, sploh zato, ker se le-ta pojavlja tudi desno od enačaja. Drugi, verjetnejši razlog je "zakasnjeno razmišljanje": najprej razpakiramo par v dve spremenljivki, potem pa razmislimo, kaj bi počeli z njima.

Odvečno "tipiziranje" Kot vedno videvam tole:

```
vrstica = str()

in tole
```

```
pozicija, crka = tuple(potek[pozicija])
```

```
in tole
```

```
vrstica += str(crka)
```

V vseh teh primerih je navajanje tipov nepotrebno.

Naj ugibam. Prvo zna biti, vsaj včasih, odsev deklariranja spremenljivk v statično tipiziranih jezikih. V C++, na primer, bi bilo potrebno spremenljivko `vrstica` deklarirati kot spremenljivko tipa `std::string`, takole:

```
std::string vrstica;
```

ali, v eni od možnih različic (ki bi jo uporabili predvsem, če bi želeli spremenljivki dati kakšno drugo začetno vrednost),

```
auto vrstica = string();
```

Predstavljam si, da je `vrstica = str()` nekaj, kar bi v Pythonu pogosto napisal programer, vaje C++. Seveda pa so možne tudi druge geneze.

Dobre razlage za `pozicija, crka = tuple(potek[pozicija])` nimam. `potek[pozicija]` je že terka; če pokličemo še `tuple`, le-ta ne bo naredil ničesar. Morda študenti ta klic dodajo za vsak slučaj? Naj se odvadijo. :)

Tudi klic `str` v `vrstica += str(crka)` ne bo naredil ničesar, saj je `str` že niz. Spet gre lahko za klic iz previdnosti, lahko pa gre za navado iz C-ja, zato morebitnim C-jašem nekaj (ponovno) razložimo: Python nima tipa `char`. V Pythonu je tudi posamičen znak že niz. Za klic funkcije ali uporabo operatorja, ki zahteva niz, ni potrebno pretvarjati znaka v niz, saj je znak že niz.

Dodatna naloga

Napiši program, ki sem ga moral napisati, ko sem sestavljal podatke za to nalogo. :)

Program dobi seznam vrstic `["Ana", "Berta", "Cilka", "Dani"]` in sestavi seznama `zacetki` in `potek`. Potek mora biti naključno premešan - ob vsakem poganjanju programa drugačen, vrstice se lahko začnejo in končujejo kjerkoli skratka, tako neumno kot zgoraj. Prav bodo prišle funkcije modula `random`, kot sta `randint` in `shuffle` ali pa celo `choice`.

Da dokažeš, da stvar dela, naj bo program oblikovan takole:

```
besedilo = ["Ana", "Berta", "Cilka", "Dani"]`
```

```
# Sledi dodatna naloga, ki sestavi premešani tabeli
```

```
print(zacetki)
```

```
print(potek)
```

```
# Obvezni del, ki dekodira in izpiše tako sestavljene začetke in ta potek v `besedilo`  
# (torej: povozi prvotni seznam)
```

```
print(besedilo)
```

Rešitev

S tem, kar znamo Bojni načrt je takšen: sestavimo seznam števil od 0 do toliko, kolikor črk je v vseh vrsticah skupaj. Nato ga naključno premešajmo. V takšnem vrstnem redu bomo zapisovali črke v potek. Potem gremo prek vseh vrstic in znotraj tega prek vseh črk.

```
besedilo = ["Ana", "Berta", "Cilka", "Dani"]
```

```
import random
```

```
n = 0
```

```
for vrstica in besedilo:  
    n += len(vrstica)
```

```
pozicije = list(range(n))  
random.shuffle(pozicije)
```

```
potek = [None] * n
```

```
zacetki = []
```

```
i = 0
```

```
for vrstica in besedilo:  
    poz = pozicije[i]  
    i += 1  
    zacetki.append(poz)  
    for crka in vrstica[:-1]:  
        nasl_poz = pozicije[i]  
        i += 1  
        potek[poz] = (nasl_poz, crka)  
        poz = nasl_poz  
    potek[poz] = (-1, vrstica[-1])
```

```
print("zacetki =", zacetki)
```

```
print("potek =", potek)
```

```
zacetki = [9, 14, 0, 10]
```

```
potek = [(12, 'C'), (-1, 'i'), (7, 'a'), (13, 'n'), (16, 'r'), (-1, 'a'), (8, 'k'), (1, 'n')]
```

potek je v začetku prazen seznam z ustreznim številom elementov.

Indeks i preprosto "potuje" po seznamu pozicije: vsako črko dodeli na mesto

`pozicije[i]`. V notranji zanki `for` najdemo pozicijo naslednje črke in v trenutno izbrani element `potek`-a zapišemo trenutno črko in naslednjo pozicijo. Pri tem notranja zanka `for` teče le do predzadnje črke; zadnjo črko zapišemo posebej, ker je njena začetna pozicija enaka `-1`. Prav tako je, seveda, izjema začetna pozicija, ki jo zapišemo v `zacetki`, ne v `potek`.

S tem, česar še ne znamo Najprej poenostavimo računanje skupne dolžine: `map(len, besedilo)` pokliče funkcijo `len` na vsakem elementu seznama `besedilo` in `sum` to vse lepo sešteje.

Glavna izboljšava je uporaba iteratorjev. S `poz = iter(pozicije)` si pripravimo iterator prek seznama `pozicije`. `next(poz)` bo zdaj vsakič vrnil naslednji element.

Še ena manjša izboljšava: "walrus", `:=`, ki hkrati priredi vrednost in jo "vrne", tako jo lahko uporabimo kot argument.

```
import random

n = sum(map(len, besedilo))
pozicije = list(range(n))
random.shuffle(pozicije)

potek = [None] * n

zacetki = []
poz = iter(pozicije)
for vrstica in besedilo:
    zacetki.append(p := next(poz))
    for c in vrstica[:-1]:
        potek[p] = (t := next(poz), c)
        p = t
    potek[p] = (-1, vrstica[-1])

print("zacetki =", zacetki)
print("potek =", potek)

zacetki = [14, 10, 6, 11]
potek = [(1, 'l'), (8, 'k'), (-1, 'a'), (0, 'i'), (-1, 'i'), (-1, 'a'), (3, 'C'), (2, 'n'),
```

Z numpyjem

```
import numpy as np

zacetki0 = np.array([0] + np.cumsum([len(vrstica) for vrstica in besedilo]))
n = zacetki0[-1]
pozicije = np.arange(n)
random.shuffle(pozicije)
```

```

crke = np.full(n, str)
crke[pozicije] = list("".join(besedilo))

naprej = np.empty(n, dtype=int)
naprej[pozicije[:-1]] = pozicije[1:]
naprej[pozicije[zacetki0 - 1]] = -1

zacetki = [pozicije[0]] + list(pozicije[zacetki0[:-1]])
potek = list(zip(naprej, crke))

print("zacetki =", zacetki)
print("potek =", potek)

zacetki = [1, 3, 13, 2]
potek = [(12, 't'), (8, 'A'), (14, 'D'), (9, 'B'), (15, 'i'), (0, 'r'), (16, 'k'), (-1, 'i')]

```

Praktično ves program se skriva v dveh vrsticah:

- `crke[pozicije] = list("".join(besedilo))` razmeče črke glede na pozicije.
- `naprej[pozicije[:-1]] = pozicije[1:]` v vsak element seznama naprej napiše, kje je naslednji element.

Ostalo je samo birokracija.