EGZ0 2021

[2pkt.] Zadanie 1.

Szablon rozwiązania: `zad1.py`

Rozważmy słowa $x[0]x[1]\cdots x[n-1]$ oraz $y[0]y[1]\cdots y[n-1]$ składające się z małych liter alfabetu łacińskiego. Takie dwa słowa są t-anagramem (dla $t \in \{0,\ldots,n-1\}$), jeśli każdej literze pierwszego słowa można przypisać taką samą literę drugiego, znajdującą się na pozycji różniącej się o najwyżej t, tak że każda litera drugiego słowa jest przypisana dokładnie jednej literze słowa pierwszego.
Proszę zaimplementować funkcję:

```
def tanagram(x, y, t):
    ...
```

która sprawdza czy słowa x i y są t-anagramami i zwraca `True` jeśli tak a `False` w przeciwnym razie. Funkcja powinna być możliwie jak najszybsza. Proszę oszacować złożoność czasową i pamięciową użytego algorytmu.

Przykład. Słowa "kotomysz" oraz "tokmysoz" są 3-anagramami, ale nie są 2-anagramami:

```
0 1 2 3 4 5 6 7   0 1 2 3 4 5 6 7  - nr litery w słowie
2 1 0 6 3 4 5 7   2 1 0 4 5 6 3 7  - nr litery przypisanej w drugim słowie
k o t o m y s z   t o k m y s o z
```

1. dla k-chaot z BST
O(n log t)

2. sliding window

k = 3

2, 1, 4, 1, 3, 4, 7

2 + 4 + 1 =

4 + 1 + 3 =

1 + 3 + 4 =

2 + 4 + 1 = 7 = s

4 + 1 + 3 = (2 + 4 + 1) − 2 + 3 = s − 2 + 3 =
s

$t = 3$

k lo t a m y s z

t f k h s o z

a b c o(e f g h i j k

0 0 0 0 0 0 0 0 0 0 0

o t m y O(n)

2 1 1 1

[2pkt.] Zadanie 2.

Szablon rozwiązania: `zad2.py`

Robot porusza się po dwuwymiarowym labiryncie i ma dotrzeć z pocycji $A = (x_a, y_a)$ na pozycję $B = (x_b, y_b)$. Robot może wykonać następujące ruchy:

1. ruch do przodu na kolejne pole,
2. obrót o 90 stopni zgodnie z ruchem wskazówek zegara,
3. obrót o 90 stopni przeciwnie do ruchów wskazówek zegara.

Obrót zajmuje robotowi 45 sekund. W trakcie ruchu do przodu robot się rozpędza i pokonanie pierwszego pola zajmuje 60 sekund, pokonanie drugiego 40 sekund, a kolejnych po 30 sekund na pole. Wykonanie obrotu zatrzymuje robota i następujące po nim ruchy do przodu ponownie go rozpędzają. Proszę zaimplementować funkcję:

```
def robot( L, A, B):
  ...
```

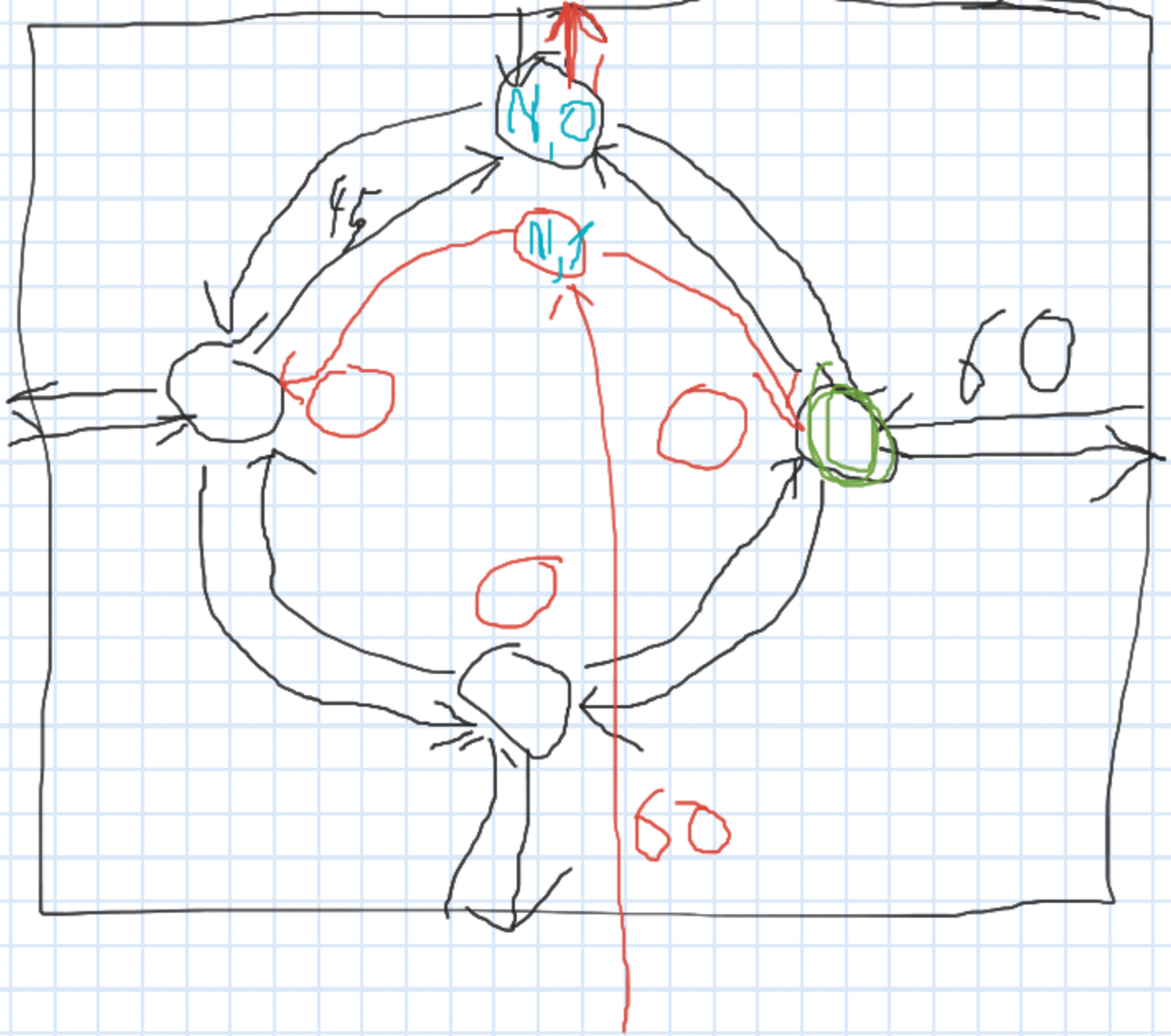
która oblicza ile minimalnie sekund robot potrzebuje na dotarcie z punktu A do punktu B (lub zwraca `None` jeśli jest to niemożliwe).

Funkcja powinna być możliwie jak najszybsza. Proszę oszacować złożoność czasową i pamięciową użytego algorytmu.

Labirynt. Labirynt reprezentowany jest przez tablicę w wierszy, z których każdy jest napisem składającym się z k kolumn. Pusty znak oznacza pole po którym robot może się poruszać, a znak `'X'` oznacza ścianę labiryntu. Labirynt zawsze otoczony jest ścianami i nie da się opuścić planszy.

Pozycja robota. Początkowo robot znajduje się na pozycji $A = (x_a, y_a)$ i jest obrócony w prawo (tj. znajduje się w wierszu y_a i kolumnie x_a, skierowany w stronę rosnących numerów kolumn).

```
      # 0123456789
L = [ "XXXXXXXXXX", # 0
      "X X      X", # 1
      "X XXXXXX X", # 2
      "X        X", # 3
      "XXXXXXXXXX", # 4
```



EGZ0 2021

[2pkt.] Zadanie 2.

Szablon rozwiązania: `zad2.py`

Dane jest drzewo binarne T, gdzie każda krawędź ma pewną wartość. Proszę zaimplementować funkcję:

```
def valuableTree(T, k):
  ...
```

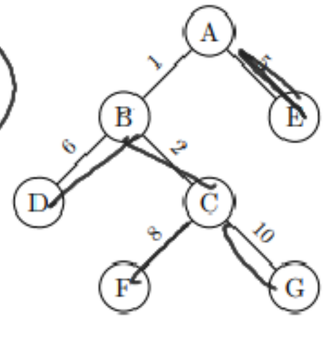
która zwraca maksymalną sumę wartości k krawędzi tworzących spójne poddrzewo drzewa T. Funkcja powinna być jak najszybsza. Proszę oszacować złożoność czasową oraz pamięciową zastosowanego algorytmu.

Drzewo T reprezentowane jest przez obiekty klasy `Node`:

```
class Node:
  def __init__(self):
    self.left     = None # lewe poddrzewo
    self.leftval  = 0    # wartość krawędzi do lewego poddrzewa jeśli istnieje
    self.right    = None # prawe poddrzewo
    self.rightval = 0    # wartość krawędzi do prawego poddrzewa jeśli istnieje
    self.X        = None # miejsce na dodatkowe dane
```

Pole X można wykorzystać do przechowywania dodatkowych informacji w trakcie obliczeń.

Przykład. Rozważmy następujące drzewo:



Wywołanie `valuableTree(A, 3)` powinno zwrócić wartość 20, odpowiadającą krawędziom B-C, C-F i C-G.

1. dynamik

$f(v, k)$ — sum max poddrzewa o kor
v i k krawędziach

$f(v, 0) = 0$

$f(v, 1) = \max(0, v.left, v.right)$

$f(l, k) = -INF$ (liść)

$1 \le j < k$

$f(v, k) = \max(f(v.left, k-1) + v.left,$
$f(v.right, k-1) + v.right,$
$f(v.left, j-1) + v.left +$
$f(v.right, k-j-1) + v.right$

$O(n \cdot k \cdot k)$
$O(nk^2)$

Szablon rozwiązania:	`egz2b.py`
Złożoność akceptowalna (1.5pkt):	$O(n^2)$, gdzie n to komnat.
Złożoność wzorcowa (+2.5pkt):	$O(n)$, gdzie n to liczba komnat.

Magiczny Wojownik obudził się w komnacie 0 pewnej tajemniczej jaskini, mając w głowie jedynie instrukcje, jakie otrzymał od Złego Maga. Wie, że komnaty są ponumerowane od 0 do $n-1$ i w każdej komnacie znajduje się troje drzwi, z których każde pozwala przejść do komnaty o wyższym numerze (cofnięcie się do komnaty o niższym numerze grozi śmiercią Wojownika; co więcej, niektóre drzwi są zablokowane) oraz skrzynia z pewną liczbą sztabek złota. Wstępnie wszystkie drzwi są zamknięte, ale jeśli w skrzyni zostanie umieszczona odpowiednia liczba sztabek złota, to drzwi się otwierają i można nimi przejść. Z każdej skrzyni można zabrać najwyżej 10 sztabek złota, ale można też w niej zostawić dowolnie wiele sztabek. Na początku Wojownik nie ma ani jednej sztabki a jego celem (na zlecenie Złego Maga) jest dojść do komnaty $n-1$ mając jak najwięcej sztabek.

Zadanie polega na zaimplementowaniu funkcji:

```
def magic( C )
```

która otrzymuje na wejściu tablicę C opisującą jaskinię ($n = |C|$) i zwraca największą liczbę sztabek złota, z którymi Wojownik może dojść do komnaty $n-1$, lub -1 jeśli dotarcie do tej komnaty jest niemożliwe. Opis jaskini jest postaci $C = [R_0, \ldots, R_{n-1}]$, gdzie każde R_i to opis komnaty postaci:

$$[G, [K_0, W_0], [K_1, W_1], [K_2, W_2]]$$

gdzie G to liczba sztabek złota w skrzyni a każda para $[K_i, W_i]$ składa się z liczby K_i sztabek słota potrzebnych do otwarcia drzwi numer i prowadzących do komnaty W_i (gdzie $W_i > i$ lub $W_i = -1$ jeśli za tymi drzwiami jest lita skała i nie da się nimi przejść nawet jeśli się je otworzy). Funkcja powinna być możliwie jak najszybsza.

Przykład. Rozważmy następującą jaskinię:

```
C = [ [8,  [ 6, 3], [ 4, 2], [7, 1]],   # 0
      [22, [12, 2], [21, 3], [0,-1]],   # 1
      [9,  [11, 3], [ 0,-1], [7,-1]],   # 2
      [15, [ 0,-1], [ 1,-1], [0,-1]] ]  # 3
```

Optymalna trasa wojownika to:

1. Wziąć 1 sztabkę złota w komnacie 0 i przejść do komnaty 1.
2. Wziąć 10 sztabek złota w komnacie 1 i przejść do komnaty 2.
3. Zostawić 2 sztabki złota w komnacie 2 i przejść do komnaty 3.

Dzięki temu na końcu wędrówki Wojownik ma 9 sztabek złota.

T[i]

f(0,0) = max(f(3,2), f(2,4), f(n-1, g) = g, f(1,1))

f(i, g) = ile najwięcej złota jest w stanie zebrać

Szablon rozwiązania:	`egz3b.py`
Złożoność akceptowalna (1.5pkt):	$O(n^3)$, gdzie n to rozmiar labiryntu.
Złożoność wzorcowa (+2.5pkt):	$O(n^2)$, gdzie n to rozmiar labiryntu.

Magiczny Wojownik który poprzednie nie dotarł do ostatniej komnaty z maksymalną liczba sztabek złota otrzymał od Dobrego Maga jeszcze jedną szansę. Musi przejść przez kwadratowy labirynt złożony z $N \times N$ komnat. Rozpoczyna wędrówkę w komnacie o współrzędnych $(0, 0)$ znajdującej się na planie w lewym górnym rogu i musi dotrzeć do komnaty o współrzędnych $(n-1, n-1)$ w prawym dolnym rogu. Niektóre komnaty (zaznaczone na planie znakiem #) są niedostępne i nie można do nich się dostać. Wojownikowi wolno poruszać się tylko w trzech kierunkach, opisanych na planie jako Góra, Prawo i Dół oraz nie wolno mu wrócić do komnaty w której już był. Zadanie postawione przez Maga polega na odwiedzeniu jak największej liczby komnat. Zadanie polega na zaimplementowaniu funkcji:

```
def maze ( L )
```

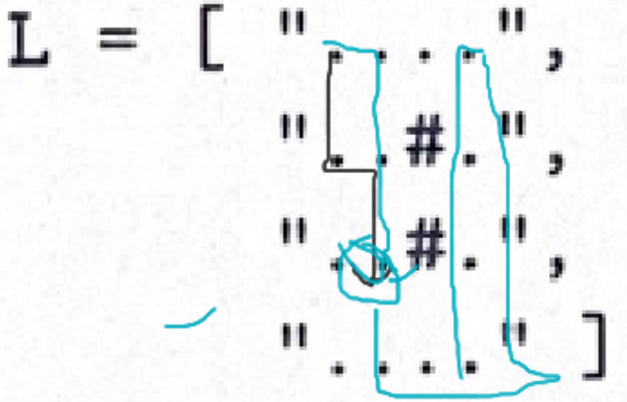
która otrzymuje na wejściu tablicę L opisującą labirynt i zwraca największą liczbę komnat, które może odwiedzić Wojownik na swojej drodze lub -1 jeśli dotarcie do końca drogi jest niemożliwe. Komnaty początkowej nie liczymy jako odwiedzonej. Funkcja powinna być możliwie jak najszybsza.

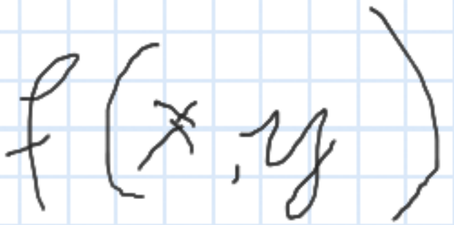
Labirynt opisuje lista $L = [W_0, W_1, W_2, \ldots, W_{n-1}]$, gdzie każde W_i to napis o długości n znaków. Znak kropki '.' oznacza dostępną komnatę a znak '#'oznacza komnatę niedostępną.

Przykład. Rozważmy następujący labirynt:

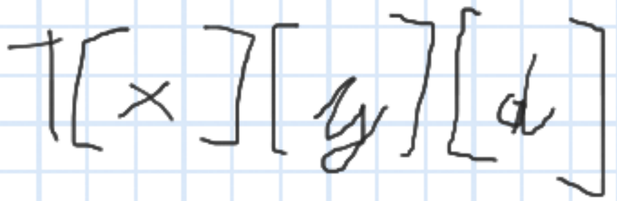
```
L = [ "....",
      "..#.",
      "..#.",
      "...." ]
```

Optymalna droga wojownika to: DDDPGGGPPDDD, podczas której Wojownik odwiedził 12 komnat.





przez ile maksymalnie komnat można z (x, y) przejść do końca



[2pkt.] Zadanie 3. EGZ0 2021

Szablon rozwiązania: zad3.py

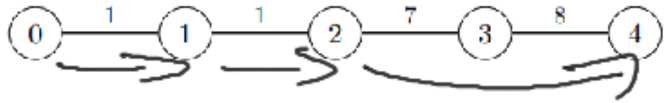
Dany jest ważony, nieskierowany graf G oraz *dwumilowe buty* - specjalny sposób poruszania się po grafie. *Dwumilowe buty* umożliwiają pokonywanie ścieżki złożonej z dwóch krawędzi grafu tak, jakby była ona pojedynczą krawędzią o wadze równej maksimum wag obu krawędzi ze ścieżki. Istnieje jednak ograniczenie - pomiędzy każdymi dwoma użyciami *dwumilowych butów* należy przejść w grafie co najmniej jedną krawędź w sposób zwyczajny. Macierz G zawiera wagi krawędzi w grafie, będące liczbami naturalnymi, wartość 0 oznacza brak krawędzi.
Proszę opisać, zaimplementować i oszacować złożoność algorytmu znajdowania najkrótszej ścieżki w grafie z wykorzystaniem mechanizmu *dwumilowych butów*.
Rozwiązanie należy zaimplementować w postaci funkcji:

```
def jumper(G, s, w):
  ...
```

która zwraca długość najkrótszej ścieżki w grafie G pomiędzy wierzchołkami s i w, zgodnie z zasadami używania *dwumilowych butów*.

Zaimplementowana funkcja powinna być możliwie jak najszybsza. Proszę przedstawić złożoność czasową oraz pamięciową użytego algorytmu.

Przykład. Rozważmy następujący graf:



Najkrótszą ścieżką między wierzchołkami 0 i 4 wykorzystującą *dwumilowe buty* będzie ścieżka $[0, 1, 2, 4]$ o długości 10 (z krawędzią $(2, 4)$ będącą *dwumilowym skokiem*). Ścieżka $[0, 2, 4]$ złożona z dwóch *dwumilowych skoków* byłaby krótsza, ale nie spełnia warunków zadania.

$O(V^2)$

A wszystkim skokiem

B wszystkim normalnie

C skoczę przez