[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
```

[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):
  ...
```

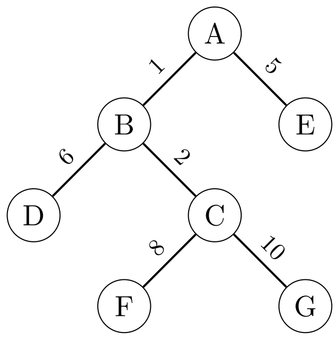
która zwraca maksymalną sumę wartości k krawędzi tworzących spójne poddrzewo drzewa T. Funkcja powinna być jak najszybsza. Proszę oszacować złożoność czasową oraz pamięciową zastosowanego algorytmu.

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.

[2pkt.] Zadanie 3.

Szablon rozwiązania: `zad3.py`

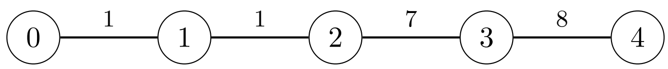
Dany jest ważony, nieskierowany graf G oraz *dwumilowe buty* - specjalny sposób poruszania się po grafie. *Dwumilowe buty* umożliwiają pokonywanie ścieżki złożonej z dwóch krawędzi grafu tak, jakby była ona pojedynczą krawędzią o wadze równej maksimum wag obu krawędzi ze ścieżki. Istnieje jednak ograniczenie - 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.