[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):
  ...
```

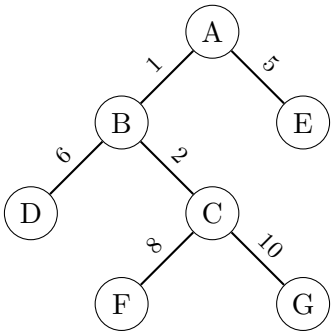
która zwraca maksymalną sumę wartości k krawędzi tworzących spójne poddrzewo drzewa T. Funkcja powinna być jak najszybsza. Proszę oszacować złożoność czasową oraz pamięciową zastosowanego algorytmu.

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`

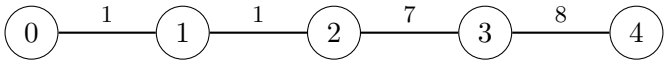
Dany jest ważony, nieskierowany graf G oraz *dwumilowe buty* - specjalny sposób poruszania się po grafie. *Dwumilowe buty* umożliwiają pokonywanie ścieżki złożonej z dwóch krawędzi grafu tak, jakby była ona pojedynczą krawędzią o wadze równej maksimum wag obu krawędzi ze ścieżki. Istnieje jednak ograniczenie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

Algorytmy i Struktury Danych
Egzamin 1 (30.VI 2021)

Format rozwiązań

Rozwiązanie każdego zadania musi się składać z **krótkiego** opisu algorytmu (wraz z uzasadnieniem poprawności i oszacowaniem złożoności obliczeniowej) oraz jego implementacji. Zarówno opis algorytmu jak i implementacja powinny się znajdować w tym samym pliku Pythona (rozszerzenie `.py`). Opis powinien być na początku pliku w formie komentarza (w pierwszej linii w komentarzu powinno być imię i nazwisko studenta). Opis nie musi być długi—wystarczy kilka zdań, jasno opisujących ideę algorytmu. Implementacja musi być zgodna z szablonem kodu źródłowego dostarczonym wraz z zadaniem. Niedopuszczalne jest w szczególności:

1. zmienianie nazwy funkcji implementującej algorytm, listy jej argumentów, lub nazwy pliku z rozwiązaniem,
2. modyfikowanie testów dostarczonych wraz z szablonem,
3. wypisywanie na ekranie jakichkolwiek napisów innych niż wypisywane przez dostarczony kod (ew. napisy dodane na potrzeby diagnozowania błędów należy usunąć przed wysłaniem zadania),
4. korzystanie z zaawansowanych struktur danych (np. słowników czy zbiorów).

Dopuszczalne jest natomiast:

1. korzystanie z następujących elementarnych struktur danych: krotka, lista, kolejka `collections.deque`, kolejka priorytetowa (`queue.PriorityQueue`),
2. korzystanie ze struktur danych dostarczonych razem z zadaniem (jeśli takie są).
3. korzystanie z wbudowanych funkcji sortujących (można założyć, że mają złożoność $O(n \log n)$). **Z wbudowanych funkcji sortowania nie wolno korzystać w zadaniu 1!**

Wszystkie inne algorytmy lub struktury danych wymagają implementacji przez studenta. Dopuszczalne jest oczywiście implementowanie dodatkowych funkcji pomocniczych w pliku z szablonem rozwiązania.

Zadania niezgodne z powyższymi ograniczeniami otrzymają ocenę 0 punktów. Rozwiązania w innych formatach (np. `.PDF`, `.DOC`, `.PNG`, `.JPG`) z definicji nie będą sprawdzane i otrzymają ocenę 0 punktów, nawet jeśli będą poprawne.

Proszę pamiętać, że rozwiązania trochę wolniejsze niż oczekiwane, ale poprawne, mają szanse na otrzymanie 1 punktu. Rozwiązania szybkie ale błędne otrzymają 0 punktów.

Testowanie rozwiązań

Żeby przetestować rozwiązania zadań należy wykonać:

```
python3 zad1.py

python3 zad2.py

python3 zad3.py
```

[2pkt.] Zadanie 1.

Szablon rozwiązania: `zad1.py`
W tym zadaniu nie wolno korzystać z wbudowanych funkcji sortowania!

Mówimy, że tablica `T` ma współczynnik nieuporządkowania równy k (jest k-Chaotyczna), jeśli spełnione są łącznie dwa warunki:

1. tablicę można posortować niemalejąco przenosząc każdy element `A[i]` o co najwyżej k pozycji (po posortowaniu znajduje się on na pozycji różniącej się od i co najwyżej o k),
2. tablicy nie da się posortować niemalejąco przenosząc każdy element o mniej niż k pozycji.

Proszę zaproponować i zaimplementować algorytm, który otrzymuje na wejściu tablicę liczb rzeczywistych `T` i zwraca jej współczynnik nieuporządkowania. Algorytm powinien być jak najszybszy oraz używać jak najmniej pamięci. Proszę uzasadnić jego poprawność i oszacować złożoność obliczeniową. Algorytm należy zaimplementować jako funkcję:

```
def chaos_index( T ):
   ...
```

przyjmującą tablicę `T` i zwracającą liczbę całkowitą będącą wyznaczonym współczynnikiem nieuporządkowania.

Przykład. Dla tablicy:

```
T = [0, 2, 1.1, 2]
```

prawidłowym wynikiem jest $k = 1$.

[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).

Przykład. Rozważmy labirynt składający się z 5 wierszy i 10 kolumn:

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

Robot ma przejść z punktu $A = (1, 1)$ do punktu $B = (8, 3)$ (czyli z wiersza nr jeden i kolumny nr jeden do wiersza nr trzy i kolumny nr osiem). Rozwiązanie wymaga następujących kroków:

1. obrót zgodnie z ruchem wskazówek zegara (45s.);
2. ruch do przodu (60s.) i drugi ruch do przodu (40s.) (robot znajuje się na polu $(1, 3)$);
3. obrót przeciwny do ruchu wskazówek zegara (45s.);
4. ruch do przodu (60s.) i kolejny (40s.) (robot znajuje się na pozycji $(3, 3)$
5. pięć kolejnych ruchów do przodu (30s. każdy).

Cały przejazd trwa 440 sekund.

[2pkt.] Zadanie 3.

Szablon rozwiązania: `zad3.py`

Dany jest zbiór przedziałów $A = \{(a_0, b_0), \ldots, (a_{n-1}, b_{n-1})\}$. Proszę zaimplementować funkcję:

```
def kintersect( A, k ):
   ...
```

która wyznacza k przedziałów, których przecięcie jest jak najdłuższym przedziałem. Zbiór A jest reprezentowany jako lista par. Końce przedziałów to liczby całkowite. Można założyć, że $k \geq 1$ oraz k jest mniejsze lub równe łącznej liczbie przedziałów w A. Funkcja powinna zwracać listę numerów przedziałów, które należą do rozwiązania.

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

Przykład: Rozważmy listę przedziałóœ:

```
A = [(0,4),(1,10),(6,7), (2,8)]
```

Dla $k = 3$ wynikiem powinno być `[0,1,3]` (lub dowolna permutacja tej listy), co daje przedziały o przecięciu $[2, 4]$, o długości $4 - 2 = 2$.

Algorytmy i Struktury Danych

Egzamin 2, 2021r.

[6pkt.] Zadanie 1.

Szablon rozwiązania:	`zad1.py`
Pierwszy próg złożoności:	$O(n^2)$
Drugi próg złożności:	$O(n \log n)$

Dany jest zbiór przedziałów domkniętych $I = \{[a_1, b_1], ..., [a_n, b_n]\}$ gdzie każdy przedział zaczyna się i kończy na liczbie naturalnej (wliczając 0). Dane są także dwie liczby naturalne x i y. Dwa przedziały można skleić (czyli zamienić na przedział będący ich sumą mnogościową) jeśli mają dokładnie jeden punkt wspólny. Jeśli pewne przedziały można posklejać tak, że powstaje z nich przedział $[x, y]$ to mówimy, że są przydatne. Proszę napisać funkcję:

```
def intuse( I, x, y )
```

która zwraca listę numerów wszystkich przydatnych przedziałów. Zbiór I jest reprezentowana [sic] jako lista par opisujących przedziały. Proszę oszacować złożoność czasową i pamięciową użytego algorytmu.

Przykład. Dla danych:

```
I = [ (3,4), (2,5), (1,3), (4,6), (1,4) ]
#       0      1      2      3      4
x = 1
y = 6
```

prawidłowym wynikiem wywołania `intuse(I, x, y)` jest dowolna permutacja listy `[0,2,3,4]`.

[6pkt.] Zadanie 3.

Szablon rozwiązania:	`zad3.py`
Pierwszy próg złożoności:	$O(n + T)$, gdzie T to łączna liczba wciśnięć przełączników; na potrzeby analizy należy przyjąć, że $T = \Omega(m \log n)$
Drugi próg złożności:	$O(m \log n)$

Dane są lampki o numerach od 0 do $n - 1$. Każda z nich może świecić na zielono, czerwono, lub niebiesko i ma jeden przełącznik, który zmienia jej kolor (z zielonego na czerwony, z czerwonego na niebieski i z niebieskiego na zielony). Początkowo wszystkie lampki świecą na zielono. Operacja (a, b) oznacza „wciśnięcie przełącznika na każdej z lampek o numerach od a do b". Wykonanych będzie m operacji. Proszę napisać funkcję:

```
def lamps( n, L )
```

która mając daną liczbę n lampek oraz listę L operacji (wykonywanych w podanej kolejności) zwraca ile maksymalnie lampek świeciło się na niebiesko (lampki są liczone na początku i po wykonaniu każdej operacji)

Przykład. Wywołanie:

```
lamps( 8, [(0,4),(2,6)] )
```

powinno zwrócić liczbę 3. Początkowo wszystkie lampki (o numerach od 0 do 7) świecą się na zielono. Następnie lampki o numerach od 0 do 4 zmieniają kolor na czerwony. Po ostatniej operacji lampki o numerach od 2 do 4 zmieniają kolor na niebieski, a lampki 5 i 6 zmieniają kolor na czerwony.

[6pkt.] Zadanie 2.

Szablon rozwiązania:	`zad2.py`
Pierwszy próg złożoności:	$O(n^2)$
Drugi próg złożności:	$O(n)$

Dane jest drzewo T zawierające n wierzchołków. Każda krawędź e drzewa ma wagę $w(e) \in \mathbb{N}$ oraz unikalny identyfikator $id(e) \in \mathbb{N}$. Wagą drzewa jest sumę [sic] wag jego krawędzi. Proszę napisać funkcję:

```
def balance( T ):
  ...
```

która zwraca identyfikator takiej krawędzi e drzewa, że usunięcie e dzieli drzewo na takie dwa, których różnica wag jest minimalna. Proszę oszacować złożoność czasową i pamięciową użytego algorytmu.

Reprezentacja drzewa. Drzewo reprezentowane jest przy pomocy węzłów typu `Node`:

```
class Node:
  def __init__( self ):    # stwórz węzeł drzewa
    self.edges   = []      # lista węzłów do których są krawędzie
    self.weights = []      # lista wag krawędzi
    self.ids     = []      # lista identyfikatorów krawędzi

  def addEdge( self, x, w, id ): # dodaj krawędź z tego węzła do węzła x
    self.edges.append( x )       # o wadze w i identyfikatorze id
    self.weights.append( w )
    self.ids.append( id )
```

Pole `edges` zawiera listę obiektów typu `Node`. Pola `edges`, `weights` oraz `ids` to listy równej długości. Należy założyć, że drzewo ma conajmniej [sic] jedną krawędź. Dopuszczalne jest dopisywanie własnych pól do `Node`.

Przykład. Rozważmy poniższe drzewo:

```
A = Node()
B = Node()
C = Node()
D = Node()
E = Node()
A.addEdge(B, 6 , 1 )
A.addEdge(C, 10, 2 )
B.addEdge(D, 5 , 3 )
B.addEdge(E, 4 , 4 )
```

Wywołanie `balance(A)` powinno zwrócić liczbę 1, czyli identyfikator krawędzi z węzła A do B o wadze 6. Usunięcie jej dzieli nasze drzewo na dwie części, o wagach 10 (krawędź z A do C) oraz 9 (drzewo z korzeniem B i krawędziamy [sic] do D i E o wagach 4 i 5).

Algorytmy i Struktury Danych

Egzamin 3, 2021r.

[6pkt.] Zadanie 1.

Szablon rozwiązania:	`zad1.py`
Pierwszy próg złożoności:	$O(n^2)$
Drugi próg złożności:	$O(n \log n)$

Mówimy, że ciąg liczb jest typu MR jeśli najpierw jest ściśle malejący a potem ściśle rosnący, albo jeśli jest tylko ściśle malejący lub tylko ściśle rosnący. Proszę zaimplementować funkcję:

```
def mr( X )
```

która mając na wejściu ciąg liczb $X = [x_0, ..., x_{n-1}]$ zwraca jeden z jego najdłuższych podciągów typu MR.

Przykład. Dla wejścia $[4, 10, 5, 1, 8, 2, 3, 4]$ wynikiem jest $[10, 5, 1, 2, 3, 4]$. Dla wejścia $[1, 10, 5]$ poprawnymi wynikami są zarówno $[1, 10]$, $[1, 5]$, jak i $[10, 5]$.

[6pkt.] Zadanie 3.

Szablon rozwiązania:	`zad3.py`
Pierwszy próg złożoności:	złożoność czasowa $O(n + m)$
Drugi próg złożności:	złożoność czasowa $O(m \log n)$ oraz pamięciowa $O(1)$

Dane jest pełne drzewo binarne T zawierające n wierzchołków. Każdy węzeł drzewa zawiera klucz będący liczbą całkowitą. Węzły drzewa numerujemy kolejnymi liczbami naturalnymi w ten sposób, że korzeń ma nr 1, jego synowie mają numery 2 i 3, a następny poziom od lewej do prawej ma numery 4,5,6,7, itd. Dany jest ciąg X zawierający m liczb naturalnych ze zbioru $\{1, ..., n\}$. Należy założyć, że m jest istotnie mniejsze niż n. Proszę zaimplementować funkcję:

```
def maxim( T, X )
```

która zwraca maksymalny klucz spośród węzłów drzewa T o numerach wymienionych w X.

Funkcja powinna być możliwie jak najszybsza - wychodząc z założenia że $m << n$, i powinna działać na stałej pamięci (poza pamięcią potrzebną na przechowywanie danych wejściowych). Proszę oszacować złożoność czasową algorytmu.

Reprezentacja drzewa. Drzewo reprezentowane jest przy pomocy węzłów typu `Node`:

```
class Node:
  def __init__( self ):
    self.left    = None  # lewe poddrzewo
    self.right   = None  # prawe poddrzewo
    self.parent  = None  # rodzic drzewa jeśli istnieje
    self.key     = None  # klucz
```

Przykład. Rozważmy drzewo, w którym klucze warstwami drzewa są umieszczone tak:

```
   5
  2 3
1 0 8 15
```

Niech $X = [3, 6, 4]$. W takim razie funkcja `maxim` powinna zwrócić wartość 8.

[6pkt.] Zadanie 2.

Szablon rozwiązania:	`zad2.py`
Pierwszy próg złożoności:	$O(dn \log n)$, gdzie n to liczba sekwencji, zaś d to długość pojedynczej sekwencji.
Drugi próg złożności:	$O(D)$, gdzie D to sumaryczna długość wszystkich sekwencji.

Dana jest lista L parami różnych napisów składających się z symboli 0, 1. Mówimy, że pewien napis s jest fajny, jeśli jest prefiksem co najmniej dwóch napisów z L (przy czym jeśli w L znajduje się napis identyczny z s, to napis s wciąż traktujemy jako jego prefiks). Dalej, mówimy że napis s jest bardzo fajny, jeśli jest fajny a zarazem żadne jego rozszerzenie (poprzez dodanie dowolnego symbolu na końcu) nie jest napisem fajnym.

Zaproponuj, uzasadnij poprawność i zaimplementuj algorytm, który otrzymuje listę napisów L (składających się z zer i jedynek) i zwraca wszystkie bardzo fajne napisy dla tej listy. Algorytm powinien być zaimplementowany jako funkcja postaci:

```
def double_prefix(L):
    ...
```

gdzie L to lista zawierająca wejściowe napisy (jako napisy w języku Python). Funkcja powinna zwrócić listę prefiksów spełniających warunki zadania (również jako listę napisów języka Python). Prefiksy można zwrócić w dowolnej kolejności.

Przykład. Dla wejścia `['0100', '0110', '1010', '1']` prawidłowym wynikiem jest dowolna permutacja listy: `['01', '1']`.

Napisy w języku Python. Python umożliwia łatwe iterowanie po elementach napisu:

```
s = '0111'
for znak in s:
  print(znak)
```

Równie łatwo można dodawać znaki do napisu:

```
s = '0111'
s += '0'
inny_s = '' + '1'
```