Algorytmy i Struktury Danych
Kolokwium 1 (23. IV 2020)

Format rozwiązań

Rozwiązanie każdego zadania powinno być wysłane jako pojedynczy plik tekstowy (rozszerzenie txt). W pierwszej linijce należy podać swoje imię i nazwisko. Rozwiązania w innych formatach (np. PDF, DOC, PNG, JPG) z definicji nie będą sprawdzane i otrzymają ocenę 0 pkt. nawet jeśli będą poprawne. W szczególnośći niedopuszczalne jest rozwiązanie zadania na kartce i wysłanie jej zdjęcia.

Python

Kod w Pythonie powienien korzystać tylko z elementarnych fragmentów języka. Niedopuszczalne jest korzystanie z wbudowanych funkcji sortujących, słowników, zbiorów, jakichkolwiek operacji import. Wszystkie użyte struktury danych należy zaimplementować samodzielnie (poza listami Pythonowymi traktowanymi jako tablice; np. `A = [0]*n` jest dopuszczalne).

Zadania

[**2pkt.**] **Zadanie 1.** Cyfra jednokrotna to taka, która występuje w danej liczbie dokładnie jeden raz. Cyfra wielokrotna to taka, która w liczbie występuje więcej niż jeden raz. Mówimy, że liczba naturalna A jest ładniejsza od liczby naturalnej B jeżeli w liczbie A występuje więcej cyfr jednokrotnych niż w B, a jeżeli cyfr jednokrotnych jest tyle samo to ładniejsza jest ta liczba, która posiada mniej cyfr wielokrotnych. Na przykład: liczba 123 jest ładniejsza od 455, liczba 1266 jest ładniejsza od 114577, a liczby 2344 i 67333 są jednakowo ładne.
Dana jest tablica T zawierająca liczby naturalne. Proszę zaimplementować funkcję:

```
pretty_sort(T)
```

która sortuje elementy tablicy T od najładniejszych do najmniej ładnych. Użyty algorytm powinien być możliwie jak najszybszy. Proszę w rozwiązaniu umieścić 1-2 zdaniowy opis algorytmu oraz proszę oszacować jego złożoność czasową.

[**2pkt.**] **Zadanie 2.** Mamy n żołnierzy różnego wzrostu i nieuporządkowaną tablicę, w której podano wzrosty żołnierzy. Żołnierze zostaną ustawieni na placu w szeregu malejąco względem wzrostu. Proszę zaimplementować funkcję:

```
section(T,p,q)
```

która zwróci tablicę ze wzrostami żołnierzy na pozycjach od p do q włącznie. Użyty algorytm powinien być możliwie jak najszybszy. Proszę w rozwiązaniu umieścić 1-2 zdaniowy opis algorytmu oraz proszę oszacować jego złożoność czasową.

[**2pkt.**] **Zadanie 3.** Proszę zaproponować algorytm, który dla tablicy liczb całkowitych rozstrzyga czy każda liczba z tablicy jest sumą dwóch innych liczb z tablicy. Zaproponowany algorytm powinien być możliwie jak najszybszy. Proszę oszacować jego złożoność obliczeniową.

Algorytmy i Struktury Danych
Kolokwium 1 (23. IV 2020)

Format rozwiązań

Rozwiązania zadań opisowych powinny być wysłane jako pojedyncze pliki tekstowe (rozszerzenie `.txt`). Rozwiązania zadań implementacyjnych powinny być wysłane jako pliki źródłowe Pythona (rozszerzenie `.py`). Krótkie wyjaśnienia i oszacowanie złożoności w zadaniach implementacyjnych powinny być w formie komentarza Pythonowego. W pierwszej linijce każdego rozwiązania należy podać swoje imię i nazwisko. Rozwiązania w innych formatach (np. `.PDF`, `.DOC`, `.PNG`, `.JPG`) z definicji nie będą sprawdzane i otrzymają ocenę 0 pkt. nawet jeśli będą poprawne. W szczególnośći niedopuszczalne jest rozwiązanie zadania na kartce i wysłanie jej zdjęcia.

Python

Kod w Pythonie powienien korzystać tylko z elementarnych fragmentów języka. Wolno korzystać z wewnętrznych funkcji sortujących. Wszystkie użyte struktury danych należy zaimplementować samodzielnie (poza listami Pythonowymi traktowanymi jako tablice; np. `A = [0]*n` jest dopuszczalne).

Zadania

[**2pkt.**] **Zadanie 1.** Proszę zaimplementować funkcję `heavy_path(T)`, która na wejściu otrzymuje drzewo `T` z ważonymi krawędziami (wagi to liczby całkowite—mogą być zarówno dodatnie, ujemne, jak i o wartości zero) i zwraca długość (wagę) najdłuższej ścieżki prostej w tym drzewie. Drzewo reprezentowane jest za pomocą obiektów typu `Node`:

```
class Node:
  def __init__( self ):
    self.children = 0   # liczba dzieci węzła
    self.child = []     # lista par (dziecko, waga krawędzi)
    ...                 # wolno dopisać własne pola
```

Poniższy kod tworzy drzewo z korzeniem `A`, który ma dwoje dzieci, węzły `B` i `C`, do których prowadzą krawędzie o wagach 5 i −1:

```
A = Node()
B = Node()
C = Node()
A.children = 2
A.child = [ (B,5), (C,-1) ]
```

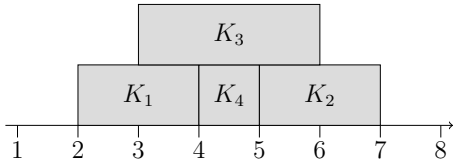
Rozwiązaniem dla drzewa `A` jest 5 (osiągnięte przez ścieżkę `A-B`; ścieżka `B-A-C` ma wagę $5 - 1 = 4$. Proszę skrótowo wyjaśnić ideę algorytmu oraz oszacować jego złożoność czasową.

[2pkt.] Zadanie 2. Algocja leży na wielkiej pustyni i składa się z miast oraz oaz połączonych drogami. Każde miasto jest otoczone murem i ma tylko dwie bramy—północną i południową. Z każdej bramy prowadzi dokładnie jedna droga do jednej oazy (ale do danej oazy może dochodzić dowolnie wiele dróg; oazy mogą też być połączone drogami między sobą). Prawo Algocji wymaga, że jeśli ktoś wjechał do miasta jedną bramą, to musi go opuścić drugą. Szach Algocji postanowił wysłać gońca, który w każdym mieście kraju odczyta zakaz formułowania zadań "o szachownicy" (obraza majestatu). Szach chce, żeby goniec odwiedził każde miasto dokładnie raz (ale nie ma ograniczeń na to ile razy odwiedzi każdą z oaz). Goniec wyjeżdża ze stolicji Algocji, miasta x, i po odwiedzeniu wszystkich miast ma do niej wrócić. Proszę przedstawić (bez implementacji) algorytm, który stwierdza czy odpowiednia trasa gońca istnieje. Proszę uzasadnić poprawność algorytmu oraz oszacować jego złożoność czasową.

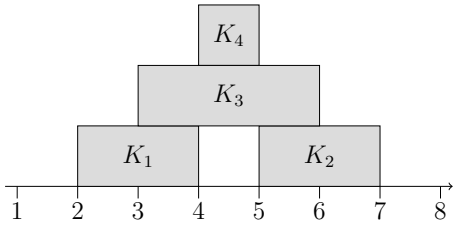
[2pkt.] Zadanie 3. Dany jest zbiór klocków $\mathcal{K} = \{K_1, \ldots, K_n\}$. Każdy klocek K_i opisany jest jako jednostronnie domknięty przedział $(a_i, b_i]$, gdzie $a_i, b_i \in \mathbb{N}$, i ma wysokość 1 (należy założyć, że żadne dwa klocki nie zaczynają się w tym samym punkcie, czyli wartości a_i są parami różne). Klocki są zrzucane na oś liczbową w pewnej kolejności. Gdy tylko spadający klocek dotyka innego klocka (powierzchnią poziomą), to jest do niego trwale doczepiany i przestaje spadać. Kolejność spadania klocków jest *poprawna* jeśli każdy klocek albo w całości ląduje na osi liczbowej, albo w całości ląduje na innych klockach. Rozważmy przykładowy zbiór klocków $\mathcal{K} = \{K_1, K_2, K_3, K_4\}$, gdzie:

$$K_1 = (2, 4], \quad K_2 = (5, 7], \quad K_3 = (3, 6], \quad K_4 = (4, 5].$$

Kolejność K_1, K_4, K_2, K_3 jest poprawna (choć są też inne poprawne kolejności) podczas gdy kolejność K_1, K_2, K_3, K_4 poprawna nie jest (K_3 nie leży w całości na innych klockach):



(a) Klocki po tym, jak spadały w poprawnej kolejności K_1, K_4, K_2, K_3.



(b) Klocki po tym, jak spadały w niepoprawnej kolejności K_1, K_2, K_3, K_4.

Proszę podać algorytm (bez implementacji), który sprawdza czy dla danego zbioru klocków istnieje poprawna kolejność spadania. Proszę uzasadnić poprawność algorytmu oraz oszacować jego złożoność. Proszę także odpowiedzieć na następujące pytanie:

> Czy jeśli początki klocków nie muszą być parami różne to algorytm dalej działa poprawnie? Jeśli tak, proszę to uzasadnić. Jeśli nie, to proszę podać kontrprzykład.

Algorytmy i Struktury Danych
Kolokwium 3 (3. VI 2020)

Format rozwiązań

Rozwiązanie każdego zadania musi składać się z 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). Implementacja musi być zgodna z szablonem kodu źródłowego dostarczonym wraz z zadaniem. Nie dopuszczalne jest w szczególności:

1. zmiana nazwy funkcji implementującej algorytm lub listy jej argumentów,
2. modyfikacja 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).

Dopuszczalne jest natomiast:

1. korzystanie z następujących elementarnych struktur danych: krotka, lista, kolejka `collections.deque`,
2. korzystanie z wbudowanych algorytmów sortowania,
3. korzystanie ze struktur danych dostarczonych razem z zadaniem.

Wszystkie inne algorytmy lub struktury danych (w tym słowniki) wymagają implementacji. Dopuszczalne jest oczywiście implementowanie dodatkowych funkcji pomocniczych w pliku z szablonem rozwiązania. Jeśli ktoś zaimplementuje standardowe drzewo BST, to może w analizie zakładać, że złożoność operacji na nim jest rzędu $O(\log n)$.

Zadania niezgodne z powyższymi ograniczeniami otrzymają ocenę 0 pkt. 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 za to poprawne, mają szanse na otrzymanie 1 punktu. Rozwiązania próbujące osiągnąć jak najlepszą złożoność, ale zaimplementowane błędnie otrzymają 0 punktów. Proszę mierzyć siły na zamiary!

Testowanie rozwiązań

Żeby przetestować rozwiązanie zadania 1 należy wykonać:

```
python3 zad1.py
```

Żeby przetestować rozwiązanie zadania 2 należy wykonać:

```
python3 zad2_testy.py
```

[2pkt.] Zadanie 1.

Szablon rozwiązania: `zad1.py`
Drzewo przedziałowe: `inttree.py`

Dana jest lista `I` przedziałów domkniętych $[a_1, b_1]$, $[a_2, b_2], \ldots, [a_n, b_n]$. Napisz funkcję `intervals( I )`, która oblicza dla każdego $i \in \{1, 2, \ldots, n\}$ długość najdłuższego ciągłego przedziału, który można osiągnąć sumując wybrane przedziały spośród pierwszych i przedziałów z listy. Funkcja powinna być możliwie jak najszybsza.

Przedziały reprezentowane są w postaci listy par. Funkcja powinna zwrócić listę liczb, w której i-ty element to długość poszukiwanego najdłuższego przedziału zbudowanego z pierwszych i elementów wejścia. Na przykład, dla listy odcinków:

```
[(1,3),(5,6),(4,7),(6,9)]
```

rozwiązaniem jest lista `[2, 2, 3, 5]` zawierająca długości odcinków: `[1, 3]`, `[1, 3]`, `[4, 7]` oraz `[4, 9]`.

Drzewo przedziałowe

W pliku `inttree.py` została Państwu dostarczona elementarna implementacja drzewa przedziałowego, tak jak było ono opisane na wykładzie. Dostępne są następujące funkcje:

1. `tree(A)` – stwórz nowe drzewo przedziałowe; przechowywane przedziały muszą być postaci $[a, b]$, gdzie liczby a i b występują w tablicy `A`; tablica `A` musi być posortowana rosnąco i nie może zawierać powtórzeń. Funkcja zwraca korzeń drzewa `T`. Złożoność: $O(|A|)$.
2. `tree_insert(T, (a,b))` – wstaw do drzewa `T` (reprezentowanego przez korzeń) przedział $[a, b]$. Złożoność: $O(\log |A|)$.
3. `tree_remove(T, (a,b))` – usuń z drzewa `T` (reprezentowanego przez korzeń) przedział $[a, b]$ (jeśli przedziału nie było w drzewie, to nic nie robi). Złożoność: $O(\log |A|)$.
4. `tree_intersect( T, x )` – zwraca listę przedziałów z drzewa `T` (reprezentowanego przez korzeń), które zawierają punkt x (niektóre przedziały mogą występować na liście dwukrotnie). Złożoność: $O(k + \log |A|)$, gdzie k to liczba zwróconych przedziałów
5. `tree_print( T )` – funkcja pomocnicza wypisująca zawartość drzewa (proszę zajrzeć do kodu, żeby zobaczyć co wypisuje).

Nie ma obowiązku korzystać z tego drzewa—można zaimplementować własne, lub użyć innej struktury danych. Jeśli ktoś zaimplementuje **klasyczne drzewo BST** to może je analizować tak, jakby operacje na nim miały **złożoność $O(\log n)$**}.

Przykład wykorzystania drzewa przedziałowego

```
from inttree import *
T = tree([1, 2, 3, 4, 5])
tree_insert(T,(1, 4))
tree_insert(T,(2, 5))
tree_print(T)
tree_remove(T,(1, 4))
tree_print(T)
tree_insert(T,(1, 3))
print(tree_intersect(T, 3))
```

[2pkt.] Zadanie 2.

Szablon rozwiązania: `zad2.py`
Plik do uruchamiania testów: `zad2_testy.py`

Dana jest tablica haszująca o rozmiarze `N` elementów oparta o adresowanie otwarte i liniowe rozwiązywanie konfliktów (z krokiem 1; dokładny kod wstawiania do tablicy znajduje się w funkcji `insert` w pliku `zad2_testy.py`). Każdy element tablicy jest obiektem zawierającym klucz (napis) oraz pole wskazujące, czy element jest zajęty:

```
class Node:
  def __init__(self, key = None, taken = False):
    self.key = key
    self.taken = taken
```

Ponadto dana jest funkcja haszująca `h(key)` przyjmująca klucz i zwracająca indeks w tablicy (w przedziale `0` do `N-1`; w pliku `zad2.py` wpisana jest konkretna wartość `N`, ale w ogólności nie należy traktować `N` jako stałej).

Niestety, w wyniku ataku komputerowego dokładnie jeden elementy tablicy haszującej zostały zmodyfikowany poprzez zmianę wartości pola `taken` na `False` oraz pola `Klucz` na `None`. Proszę zaproponować i zaimplementować funkcję:

```
def recover(hash_tab):
  ...
```

która sprawdza, czy w tablicy haszującej przekazanej przez argument wszystkie elementy z polem `taken` równym `True` mogą być poprawnie znalezione (procedura wyszukiwania w tablicy haszującej to `find` z pliku `zad2_testy.py`). Jeżeli tak nie jest, funkcja powinna „naprawić" tablicę w taki sposób, by wszystkie elementy (w których `taken == True`) były osiągalne. W każdym przypadku funkcja powinna zwrócić tablicę (potencjalnie naprawioną) jako wynik. Funkcja może używać jedynie stałego (niezależnego od `N`) rozmiaru dodatkowej pamięci operacyjnej (a więc powinna działać w miejscu - nie można dodawać pól do elementów tablicy lub utworzyć nowej tablicy haszującej). Zaproponowane rozwiązanie powinno być możliwie jak najszybsze.

Algorytmy i Struktury Danych
Kolokwium Zaliczeniowe I (3. VII 2020)

Format rozwiązań

Rozwiązanie każdego zadania musi składać się z 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). 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).

Dopuszczalne jest natomiast:

1. korzystanie z następujących elementarnych struktur danych: krotka, lista, kolejka `collections.deque`,
2. korzystanie z wbudowanych algorytmów sortowania,
3. korzystanie ze struktur danych dostarczonych razem z zadaniem.

Wszystkie inne algorytmy lub struktury danych (w tym słowniki) wymagają implementacji. Dopuszczalne jest oczywiście implementowanie dodatkowych funkcji pomocniczych w pliku z szablonem rozwiązania. Jeśli ktoś zaimplementuje standardowe drzewo BST, to może w analizie zakładać, że złożoność operacji na nim jest rzędu $O(\log n)$.

Zadania niezgodne z powyższymi ograniczeniami otrzymają ocenę 0 pkt. 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 za to poprawne, mają szanse na otrzymanie 1 punktu. Rozwiązania próbujące osiągnąć jak najlepszą złożoność, ale zaimplementowane błędnie otrzymają 0 punktów. Proszę mierzyć siły na zamiary!

Testowanie rozwiązań

Żeby przetestować rozwiązania zadań należy wykonać:

```
python3 zad1.py

python3 zad2.py

python3 zad3.py
```

[2pkt.] Zadanie 1.

Szablon rozwiązania: `zad1.py`

Dana jest tablica dwuwymiarowa `G`, przedstawiająca macierz sąsiedztwa skierowanego grafu ważonego, który odpowiada mapie drogowej (wagi przedstawiają odległości, liczba `-1` oznacza brak krawędzi). W niektórych wierzchołkach są stacje paliw, podana jest ich lista `P`. Pełnego baku wystarczy na przejechanie odległości `d`. Wjeżdżając na stację samochód zawsze jest tankowany do pełna. Proszę zaimplemntować funkcję `jak_dojade(G, P, d, a, b)`, która szuka najkrótszej możliwej trasy od wierzchołka `a` do wierzchołka `b`, jeśli taka istnieje, i zwraca listę kolejnych odwiedzanych na trasie wierzchołków (zakładamy, że w `a` też jest stacja paliw; samochód może przejechać najwyżej odległość `d` bez tankowania).

Zaproponowana funkcja powinna być możliwe jak najszybsza. Uzasadnij jej poprawność i oszacuj złożoność obliczeniową.

Przykład Dla tablic

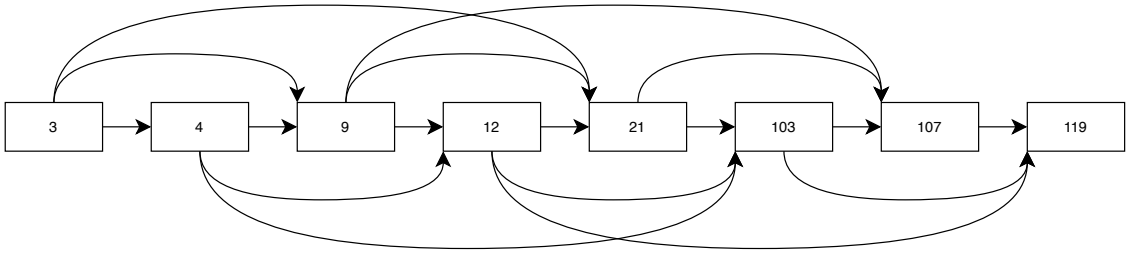
```
G = [[-1, 6,-1, 5, 2],
     [-1,-1, 1, 2,-1],
     [-1,-1,-1,-1,-1],
     [-1,-1, 4,-1,-1],
     [-1,-1, 8,-1,-1]]
P = [0,1,3]
```

funkcja `jak_dojade(G, P, 5, 0, 2)` powinna zwrócić `[0,3,2]`. Dla tych samych tablic funkcja `jak_dojade(G, P, 6, 0, 2)` powinna zwrócić `[0,1,2]`, natomiast `jak_dojade(G, P, 3, 0, 2)` powinna zwrócić `None`.

[2pkt.] Zadanie 2.

Szablon rozwiązania: `zad2.py`

W szybkiej liscie odsyłaczowej i-ty element posiada referencje (odsyłacze) do elementów: $i+2^0$, $i+2^1$, $i+2^2$, ... (lista odsyłaczy z i-tego elementu kończy się na ostatnim elemencie o numerze postaci $i+2^k$, który występuje w liście). Lista ta przechowuje liczby całkowite w kolejności niemalejącej. Przykładową szybką listę przedstawia poniższy rysunek:



Napisz funkcję `fast_list_prepend`:

```
def fast_list_prepend(L, a):
  ...
```

która przyjmuje referencję na pierwszy węzeł szybkiej listy (`L`) oraz liczbę całkowitą (`a`) mniejszą od wszystkich liczb w przekazanej liście i wstawia tę liczbę na początek szybkiej listy (jako nowy węzeł). W wyniku dodania nowego elementu powinna powstać prawidłowa szybka lista. W szczególności każdy węzeł powinien mieć poprawne odsyłacze do innych węzłów. Funkcja powinna zwrócić referencję na nowy pierwszy węzeł szybkiej listy.

Zaproponowana funkcja powinna być możliwe jak najszybsza. Uzasadnij jej poprawność i oszacuj złożoność obliczeniową. Węzły szybkiej listy reprezentowane są w postaci:

```
class FastListNode:
    def __init__(self, a):
        self.a = a     # przechowywana liczba całkowita
        self.next = [] # lista odsyłaczy do innych elementów; początkowo pusta

    def __str__(self): # zwraca zawartość węzła w postaci napisu
        res = 'a: ' + str(self.a) + '\t' + 'next keys: '
        res += str([n.a for n in self.next])
        return res
```

[2pkt.] Zadanie 3.

Szablon rozwiązania: `zad3.py`

Dana jest tablica `A` zawierająca `n = len(A)` liczb naturalnych. Dodatkowo wiadomo, że `A` w sumie zawiera `k` różnych liczb (należy założyć, że `k` jest dużo mniejsze niż `n`). Proszę zaimplementować funkcję `longest_incomplete(A, k)`, która zwraca długość najdłuższego spójnego ciągu elementów z tablicy `A`, w którym nie występuje wszystkie `k` liczb. (Można założyć, że podana wartość `k` jest zawsze prawidłowa.)

Przykład Dla tablicy

```
A = [1,100, 5, 100, 1, 5, 1, 5]
```

wartością wywołania `longest_incomplete(A, 3)` powinno być `4` (ciąg `1, 5, 1, 5` z końca tablicy).

Algorytmy i Struktury Danych
Kolokwium Zaliczeniowe II (1. IX 2020)

Format rozwiązań

Rozwiązanie każdego zadania musi składać się z 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). 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).

Dopuszczalne jest natomiast:

1. korzystanie z następujących elementarnych struktur danych: krotka, lista, kolejka `collections.deque`,
2. korzystanie z wbudowanych algorytmów sortowania (**poza pierwszym zadaniem**),
3. korzystanie ze struktur danych dostarczonych razem z zadaniem.

Wszystkie inne algorytmy lub struktury danych (w tym słowniki) wymagają implementacji. Dopuszczalne jest oczywiście implementowanie dodatkowych funkcji pomocniczych w pliku z szablonem rozwiązania. Jeśli ktoś zaimplementuje standardowe drzewo BST, to może w analizie zakładać, że złożoność operacji na nim jest rzędu $O(\log n)$.

Zadania niezgodne z powyższymi ograniczeniami otrzymają ocenę 0 pkt. 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 błędnie otrzymają 0 punktów. Proszę mierzyć siły na zamiary!

Testowanie rozwiązań

Żeby przetestować rozwiązania zadań należy wykonać:

```
python3 zad1.py

python3 zad2.py

python3 zad3.py
```

[2pkt.] Zadanie 1.

Szablon rozwiązania: `zad1.py`

Mówimy, że punkt (x, y) słabo dominuje punkt (x', y') jeśli $x \leq x'$ oraz $y \leq y'$ (w szczególności każdy punkt słabo dominuje samego siebie). Dana jest tablica P zawierająca n punktów. Proszę zaimplementować funkcję `dominance(P)`, która zwraca tablicę S taką, że:

1. elementami S są indeksy punktów z P,
2. dla każdego punktu z P, S zawiera indeks punktu, który go słabo dominuje,
3. S zawiera minimalną liczbę elementów.

Przykład. Dla tablicy:

```
P = [ (2,2), (1,1), (2.5,0.5), (3,2), (0.5,3) ]
#       0      1        2        3        4
```

wynikiem jest, między innymi:

```
S = [ 1, 4, 2 ]
```

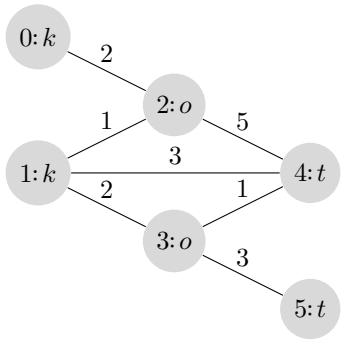
[2pkt.] Zadanie 2.

Szablon rozwiązania: `zad2.py`

Dany jest graf nieskierowany $G = (V, E)$, gdzie każdy wierzchołek z V ma przypisaną małą literę z alfabetu łacińskiego, a każda krawędź ma wagę (dodatnią liczbę całkowitą). Dane jest także słowo $W = W[0], \ldots, W[n-1]$ składające się małych liter alfabetu łacińskiego. Należy zaimplementować funkcję `letters(G,W)`, która oblicza długość najkrótszej ścieżki w grafie G, której wierzchołki układają się dokładnie w słowo W (ścieżka ta nie musi być prosta i może powtarzać wierzchołki). Jeśli takiej ścieżki nie ma, należy zwrócić -1.

Struktury danych. Graf G ma n wierzchołków ponumerowanych od 0 do $n-1$ i jest reprezentowany jako para (L, E). L to lista o długości n, gdzie $L[i]$ to litera przechowywana w wierzchołku i. E jest listą krawędzi i każdy jej element jest trójką postaci (u, v, w), gdzie u i v to wierzchołki połączone krawędzią o wadze w.

Przykład. Rozważmy graf G przedstawiony poniżej:



W reprezentacji przyjętej w zadaniu mógłby być zapisany jako:

```
    # 0   1   2   3   4   5
L = ["k","k","o","o","t","t"]
E = [(0,2,2), (1,2,1), (1,4,3), (1,3,2), (2,4,5), (3,4,1), (3,5,3) ]
G = (L,E)
```

Rozwiązaniem dla tego grafu i słowa `W = "kto"` jest 4 i jest osiągane przez ścieżkę $1 - 4 - 3$. Inna ścieżka realizująca to słowo to $1 - 4 - 2$, ale ma koszt 8.

[2pkt.] Zadanie 3.

Szablon rozwiązania: `zad3.py`

Dana jest tablica T zawierająca N liczb naturalnych. Z pozycji a można przeskoczyć na pozycję b jeżeli liczby $T[a]$ i $T[b]$ mają co najmniej jedną wspólną cyfrę. Koszt takego skoku równy $|T[a] - T[b]|$. Proszę napisać funkcję, która wyznacza minimalny sumaryczny koszt przejścia z najmniejszej do największej liczby w tablicy T. Jeżeli takie przejście jest niemożliwe, funkcja powinna zwrócić wartość -1.

Przykład Dla tablicy `T = [123,890,688,587,257,246]` wynikiem jest liczba 767, a dla tablicy `T = [587,990,257,246,668,132]` wynikiem jest liczba -1.