

Algorytmy i Struktury Danych  
Egzamin Poprawkowy 2 (15. IX 2020)

Imię i nazwisko: \_\_\_\_\_

A

### 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 (**poza zadaniem 3**),
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 szansę na otrzymanie 1 punktu. Rozwiązania szybsze, ale błędne 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

Każdy nieskierowany, spójny i acykliczny graf  $G = (V, E)$  możemy traktować jako drzewo. Korzeniem tego drzewa może być dowolny wierzchołek  $v \in V$ . Napisz funkcję `best_root(L)`, która przyjmuje nieskierowany, spójny i acykliczny graf  $G$  (reprezentowany w postaci listy sąsiedztwa) i wybiera taki jego wierzchołek, by wysokość zakorzenionego w tym wierzchołku drzewa była możliwie najmniejsza. Jeśli kilka wierzchołków spełnia warunki zadania, funkcja może zwrócić dowolny z nich. Wysokość drzewa definiujemy jako liczbę krawędzi od korzenia do najdalszego liścia. Uzasadnij poprawność zaproponowanego algorytmu i oszacuj jego złożoność obliczeniową.

Funkcja `best_root(L)` powinna zwrócić numer wierzchołka wybranego jako korzeń. Wierzchołki numerujemy od 0. Argumentem `best_root(L)` jest lista postaci:

$$L = [l_0, l_1, \dots, l_{n-1}],$$

gdzie  $l_i$  to lista zawierająca numery wierzchołków będących sąsiadami  $i$ -tego wierzchołka. Można przyjąć (bez weryfikacji), że lista opisuje graf spełniający warunki zadania. W szczególności, graf jest spójny, acykliczny, oraz jeśli  $a \in l_b$  to  $b \in l_a$  (graf jest nieskierowany). Nagłówek funkcji powinien mieć postać:

```
def best_root(L):  
    ...
```

**Przykład.** Dla listy sąsiedztwa postaci:

```
L = [ [ 2 ],  
      [ 2 ],  
      [ 0, 1, 3 ],  
      [ 2, 4 ],  
      [ 3, 5, 6 ],  
      [ 4 ],  
      [ 4 ] ]
```

funkcja powinna zwrócić wartość 3.

[2pkt.] **Zadanie 2.**

**Szablon rozwiązania:** zad2.py

Dany jest ciąg klocków  $(a_1, b_1), \dots, (a_n, b_n)$ . Każdy klocek zaczyna się na pozycji  $a_i$  i ciągnie się do pozycji  $b_i$ . Klocki mogą spadać w kolejności takiej jak w ciągu. Proszę zaimplementować funkcję `tower(A)`, która wybiera możliwie najdłuższy podciąg klocków taki, że spadając tworzą wieżę i żaden klocek nie wystaje poza którykolwiek z wcześniejszych klocków. Do funkcji przekazujemy tablicę `A` zawierającą pozycje klocków  $a_i, b_i$ . Funkcja powinna zwrócić maksymalną wysokość wieży jaką można uzyskać w klocków w tablicy `A`.

**Przykład** Dla tablicy `A = [(1,4), (0,5), (1,5), (2,6), (2,4)]` wynikiem jest 3, natomiast dla tablicy `A = [(10,15), (8,14), (1,6), (3,10), (8,11), (6,15)]` wynikiem jest 2.

### [2pkt.] Zadanie 3.

Szablon rozwiązania: zad3.py

Dana jest struktura realizująca listę jednokierunkową:

```
class Node:
    def __init__( self, val ):
        self.next = None
        self.val = val
```

Proszę napisać funkcję, która mając na wejściu ciąg tak zrealizowanych posortowanych list scala je w jedną posortowaną listę (składającą się z tych samych elementów).

**Przykład** Dla tablicy `[[0,1,2,4,5], [0,10,20], [5,15,25]]` - po przekształceniu jej elementów z Python'owskich list na listy jednokierunkowe - wynikiem powinna być lista jednokierunkowa, która po przekształceniu jej na listę Python'owską przyjmie postać `[0,0,1,2,4,5,5,10,15,20,25]`.