

Algorytm i Struktury Danych

Kolokwium 1: Zadanie A (31.III.2022)

Format rozwiązań

Rozwiązanie zadania musi się składać z krótkiego opisu algorytmu (wraz z uzasadnieniem poprawności) 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. korzystanie z wbudowanych funkcji sortujących,
2. korzystanie z zaawansowanych struktur danych (np. słowników czy zbiorów),
3. zmienianie nazwy funkcji implementującej algorytm, listy jej argumentów, lub nazwy pliku z rozwiązaniem,
4. 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,
2. korzystanie ze struktur danych dostarczonych razem z zadaniem (jeśli takie są).

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.

Testowanie rozwiązań

Żeby przetestować rozwiązanie zadania należy wykonać polecenie: `python kol1a.py`

Kilka uwag o Pythonie

```
x = "abcdefg"
y = x[::-1]      # y staje się nowym napisem "gfedcba"
z = y            # z i y wskazują na ten sam napis (koszt tego przypisania to O(1))
                 # niezależnie od długości napisu y
ord("a")         # wynikiem jest kod litery 'a' (97)
chr(97)          # wynikiem jest litera o kodzie 97 ("a")
```

| | |
|----------------------------------|---|
| Szablon rozwiązania: | <code>kol1a.py</code> |
| Pierwszy próg złożoności: | $O(N + n \log n)$, gdzie N to łączna długość napisów w tablicy wejściowej a n to liczba wyrazów. |
| Drugi próg złożoności: | $O(N \log N)$ lub $O(nk)$ gdzie N to łączna długość napisów w tablicy wejściowej, n to liczba wyrazów a k to długość najdłuższego słowa |

Mówimy, że dwa napisy są sobie równoważne, jeśli albo są identyczne, albo byłyby identyczne, gdyby jeden z nich zapisać od tyłu. Na przykład napisy "kot" oraz "tok" są sobie równoważne, podobnie jak napisy "pies" i "pies". Dana jest tablica T zawierająca n napisów o łącznej długości N (każdy napis zawiera co najmniej jeden znak, więc $N \geq n$; każdy napis składa się wyłącznie z małych liter alfabetu łacińskiego). Siłą napisu $T[i]$ jest liczba indeksów j takich, że napisy $T[i]$ oraz $T[j]$ są sobie równoważne. Napis $T[i]$ jest najsilniejszy, jeśli żaden inny napis nie ma większej siły.

Proszę zaimplementować funkcję $g(T)$, która zwraca siłę najsilniejszego napisu z tablicy T . Na przykład dla wejścia:

```
#      0      1      2      3      4      5      6
T = ["pies", "mysz", "kot", "kogut", "tok", "seip", "kot"]
```

wywołanie $g(T)$ powinno zwrócić 3. Algorytm powinien być możliwie jak najszybszy. Proszę podać złożoność czasową i pamięciową zaproponowanego algorytmu.

Szablon rozwiązania: kol1b.py

Pierwszy próg złożoności: $O(N)$, gdzie N to łączna długość napisów w tablicy wejściowej.

Drugi próg złożoności: $O(N \log N)$

Dwa słowa są anagramami jeśli składają się z dokładnie tej samej liczby tych samych liter. Na przykład anagramami są słowa „algorytm” i „logarytm”, podczas gdy słowa „katar” i „totem” anagramami nie są. Jeśli dwa słowa są anagramami, to są tej samej długości.

Dana jest tablica T składająca się z pewnej liczby słów, gdzie każde słowo składa się z małych liter alfabetu łacińskiego. Popularnością anagramową słowa $T[i]$ nazywamy liczbę takich indeksów j , że słowo $T[j]$ jest anagramem słowa $T[i]$.

Proszę zaimplementować funkcję $f(T)$, która zwraca popularność najpopularniejszego anagramu. Na przykład dla wejścia:

```
#      0      1      2      3      4      5      6      7
T = ["tygrys", "kot", "wilk", "trysyg", "wlik", "sygryt", "likw", "tygrys"]
```

wywołanie $f(T)$ powinno zwrócić liczbę 4. Algorytm powinien być możliwie jak najszybszy. Proszę podać złożoność czasową i pamięciową zaproponowanego algorytmu.

Algorytmy i Struktury Danych
Kolokwium 2: Zadanie A (12.V.2022)

Format rozwiązań

Rozwiązanie zadania musi się składać z krótkiego opisu algorytmu (wraz z uzasadnieniem poprawności) 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 zdąń, jasno opisujących ideę algorytmu. Implementacja musi być zgodna z szablonem kodu źródłowego dostarczonym wraz z zadaniem. Niedopuszczalne jest w szczególności:

1. korzystanie z zaawansowanych struktur danych (np. słowników czy zbiorów),
2. zmienianie nazwy funkcji implementującej algorytm, listy jej argumentów, lub nazwy pliku z rozwiązaniem,
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`, kolejka priorytetowa (`queue.PriorityQueue` lub `heapq`),
2. korzystanie ze struktur danych dostarczonych razem z zadaniem (jeśli takie są),
3. korzystanie z wbudowanych funkcji sortujących (można założyć, że mają złożoność $O(n \log n)$).

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.

Testowanie rozwiązań

Żeby przetestować rozwiązanie zadania należy wykonać polecenie: `python kol2a.py`

| | |
|---------------------------------------|---|
| Szablon rozwiązania: | <code>kol2a.py</code> |
| Złożoność akceptowalna (3pkt): | $O(n^2)$, gdzie n to łączna liczba punktów. |
| Złożoność wzorcowa (+1pkt): | $O(n \log n)$, gdzie n to łączna liczba punktów. |

Cieżarówka o numerze bocznym 1212 musi przejechać z punktu A do punktu B . Na trasie znajduje się pewna liczba punktów kontrolnych oraz pewna liczba punktów przesiadkowych. Ciężarówką jedzie dwóch kierowców-zmienników, Jacek i Marian. Z punktu A rusza Jacek, ale kierowcy muszą się zmieniać najpóźniej na trzecim punkcie przesiadkowym od ostatniej zmiany. Zadanie polega na takim zaplanowaniu przesiadek, żeby Marian był za kierownicą podczas mijania jak najmniejszej liczby punktów kontrolnych.

Można sobie wyobrazić, że trasa przebiega po jednowymiarowej linii, gdzie punkt A jest na pozycji 0 a punkty przesiadkowe i kontrolne mają współrzędne naturalne (większe od zera, parne różne). Proszę zaimplementować funkcję:

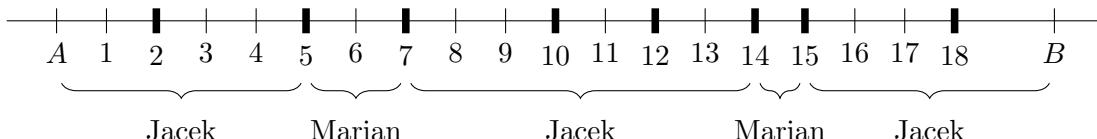
```
def drivers( P, B )
```

która na wejściu otrzymuje tablicę P zawierającą współrzędne punktów przesiadkowych i kontrolnych, oraz współrzędną punktu B . Każdy element tablicy P to para (x, t) , gdzie x to współrzędna punktu a t ma wartość `True` jeśli jest to punkt przesiadkowy oraz `False` jeśli jest to punkt kontrolny. Tablica P nie musi być posortowana. Funkcja powinna zwrócić indeksy punktów przesiadkowych, na których kierowcy zamieniają się miejscami. Funkcja powinna być możliwie jak najszybsza.

Rozważmy następujące dane:

```
p = True
c = False
#      0      1      2      3      4      5      6      7      8      9
P = [(1,c),(3,c),(4,c),(6,c),(8,c),(9,c),(11,c),(13,c),(16,c),(17,c),
      (2,p),(5,p),(7,p),(10,p),(12,p),(14,p),(15,p),(18,p)]
#      10     11     12     13     14     15     16     17
B = 20
```

wywołanie `drivers(P, B)` może zwrócić tablicę `[11, 12, 15, 16]`, odpowiadające poniższej sekwenacji zmian (grube kreski odpowiadają punktom przesiadkowym):



Podpowiedź. Proszę się zastanowić, ile punktów kontrolnych musi przejechać Marian w momencie, gdy dojeżdżamy do danego punktu przesiadkowego i dana osoba prowadzi.

Algorytmy i Struktury Danych
Kolokwium 2: Zadanie B (12.V.2022)

Format rozwiązań

Rozwiązanie zadania musi się składać z krótkiego opisu algorytmu (wraz z uzasadnieniem poprawności) 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. korzystanie z zaawansowanych struktur danych (np. słowników czy zbiorów),
2. zmienianie nazwy funkcji implementującej algorytm, listy jej argumentów, lub nazwy pliku z rozwiązaniem,
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`, kolejka priorytetowa (`queue.PriorityQueue` lub `heapq`),
2. korzystanie ze struktur danych dostarczonych razem z zadaniem (jeśli takie są),
3. korzystanie z wbudowanych funkcji sortujących (można założyć, że mają złożoność $O(n \log n)$).

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.

Testowanie rozwiązań

Żeby przetestować rozwiązanie zadania należy wykonać polecenie: `python kol2b.py`

| | |
|---------------------------------------|---|
| Szablon rozwiązania: | <code>kol2b.py</code> |
| Złożoność akceptowalna (3pkt): | $O(n^2)$, gdzie n to łączna liczba parkingów. |
| Złożoność wzorcowa (+1pkt): | $O(n \log n)$, gdzie n to łączna liczba parkingów. |

Kierowca ciężarówki przewozi towary z miasta A do miasta B . W pewnych miejscowościach trasy przejazdu znajdują się parkingi. Przejeżdżając obok parkingu kierowca może (ale nie musi) się na nim zatrzymać i odpocząć. Przepisy transportowe narzucają jednak pewne ograniczenia związane z bezpieczeństwem:

1. Maksymalna liczba kilometrów, którą można przejechać bez zatrzymania wynosi T . Od zasady tej istnieje jeden wyjątek, opisany w punkcie 2.
2. W trakcie całego przejazdu z A do B kierowca może jeden raz przekroczyć limit T kilometrów jazdy bez zatrzymania. Może wówczas przejechać nie więcej niż $2T$ kilometrów bez zatrzymania.

Niestety, parkingi na trasie są płatne. Co więcej, opłaty za postój różnią się pomiędzy parkinkami. Kierowca musi więc wybrać miejsca postoju w taki sposób, by przejechać trasę zgodnie z obowiązującymi przepisami i równocześnie zapłacić możliwie jak najmniej za postoje.

Zaproponuj i zaimplementuj algorytm, który wylicza minimalny koszt przejechania z miasta A do miasta B zgodnie z opisanymi przepisami transportu towarów. Koszt przejazdu z A do B definiujemy jako sumę opłat za parkowanie w miejscowościach, w których kierowca się zatrzymał (nie liczymy ceny paliwa; nie bierzemy pod uwagę czasu postoju na parkingu). W miastach A i B opłata nie jest pobierana. Uzasadnij poprawność zaproponowanego algorytmu i oszacuj jego złożoność obliczeniową.

Algorytm należy zaimplementować jako funkcję:

```
def min_cost( O, C, T, L )
```

Argumentami funkcji są:

- Tablica O zawierająca pozycje parkingów na trasie z A do B . $O[i]$ to liczba kilometrów (wzdłuż trasy przejazdu) od A do i -tego parkingu.
- Tablica C zawierająca ceny za postój na poszczególnych parkingach. $C[i]$ to opłata za zatrzymanie na i -tym parkingu.
- Maksymalna liczba kilometrów T , którą można przejechać bez zatrzymywania (z zastrzeżeniem wyjątku opisanego powyżej).
- Długość L trasy (liczba kilometrów od A do B wzdłuż trasy przejazdu).

Wszystkie wartości przekazane w tablicach O i C oraz argumenty T i L to dodatnie liczby naturalne. Tablice nie muszą być posortowane. Funkcja `min_cost` powinna zwrócić jedną liczbę naturalną: minimalny koszt przejazdu z A do B . Można założyć, że parkingi są tak rozmieszczone, że da się przejechać z A do B zgodnie z obowiązującymi zasadami. Funkcja powinna być możliwie jak najszybsza.

Przykład. Dla danych:

```
O = [17, 20, 11, 5, 12]
C = [9, 7, 7, 7, 3]
T = 7
L = 25
```

wywołanie `min_cost(O, C, T, L)` powinno zwrócić wartość 10.

Podpowiedź. Zastanów się, jaki jest koszt dojechania do parkingu i mogąc lub nie mogąc wykorzystać wyjątku, o którym mowa w punkcie 2.

Algorytmy i Struktury Danych
Kolokwium 3: Zadanie A (15.VI.2022)

Format rozwiązań

Rozwiązanie zadania musi się składać z krótkiego opisu algorytmu (wraz z uzasadnieniem poprawności) 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 zdąń, jasno opisujących ideę algorytmu. Implementacja musi być zgodna z szablonem kodu źródłowego dostarczonym wraz z zadaniem. Niedopuszczalne jest w szczególności:

1. korzystanie z zaawansowanych struktur danych (np. słowników czy zbiorów),
2. zmienianie nazwy funkcji implementującej algorytm, listy jej argumentów, lub nazwy pliku z rozwiązaniem,
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`, kolejka priorytetowa (`queue.PriorityQueue` lub `heapq`),
2. korzystanie ze struktur danych dostarczonych razem z zadaniem (jeśli takie są),
3. korzystanie z wbudowanych funkcji sortujących (można założyć, że mają złożoność $O(n \log n)$).

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.

Testowanie rozwiązań

Żeby przetestować rozwiązanie zadania należy wykonać polecenie: `python kol3a.py`

Szablon rozwiązania:

kol3a.py

Złożoność akceptowalna (1.5pkt):

$O(n^2)$, gdzie n to liczba planet.

Złożoność wzorcowa (+2.5pkt):

$O(m \log n)$, gdzie m to długość listy E a n to liczba planet.

Układ planetarny Algon składa się z n planet o numerach od 0 do $n - 1$. Niestety własności fizyczne układu powodują, że nie da się łatwo przelecieć między dowolnymi dwiema planetami. Na szczęście mozolna eksploracja kosmosu doprowadziła do stworzenia listy E dopuszczalnych bezpośrednich przelotów. Każdy element listy E to trójka postaci (u, v, t) , gdzie u i v to numery planet (można założyć, że $u < v$) a t to czas podróży między nimi (przelot z u do v trwa tyle samo co z v do u). Dodatkową nietypową własnością układu Algon jest to, że niektóre planety znajdują się w okolicy osobliwości. Znajdując się przy takiej planecie możliwe jest zagłębie czasoprzestrzeni umożliwiające przedostanie się do dowolnej innej planety leżącej przy osobliwości w czasie zerowym.

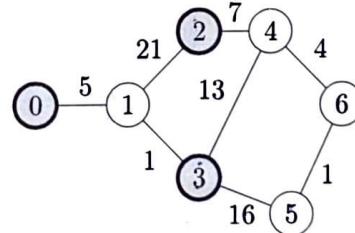
Zadanie polega na zaimplementowaniu funkcji:

```
def spacetravel( n, E, S, a, b )
```

która zwraca najkrótszy czas podróży z planety a do planety b , mając do dyspozycji listę możliwych bezpośrednich przelotów E oraz listę S planet znajdujących się koło osobliwości. Jeśli trasa nie istnieje, to funkcja powinna zwrócić `None`.

Rozważmy następujące dane:

```
E = [(0,1, 5),
      (1,2,21),
      (1,3, 1),
      (2,4, 7),
      (3,4,13),
      (3,5,16),
      (4,6, 4),
      (5,6, 1)]
S = [ 0, 2, 3 ]
a = 1
b = 5
n = 7
```



wywołanie `startravel(n, E, S, a, b)` powinno zwrócić liczbę 13. Odwiedzamy po kolej planety 1, 3, 2, 4, 6 i kończymy na planecie 5 (z planety 2 do 3 dostajemy się przez zagłębie czasoprzestrzeni). Gdyby $a = 1$ a $b = 2$ to wynikiem byłby czas przelotu 1.

Podpowiedź. Ile zagłębie czasoprzestrzeni warto maksymalnie rozważyć? A ile minimalnie?