Algorytmy i Struktury Danych
Kolokwium II (17. V 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 $O(n \log n)$).

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 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.

Testowanie rozwiązań

Żeby przetestować rozwiązania zadań należy wykonać:

```
python3 zad1.py

python3 zad2.py

python3 zad3.py
```

[2pkt.] Zadanie 1.

Szablon rozwiązania: `zad1.py`

Inwestor planuje wybudować nowe osiedle akademików. Architekci przedstawili projekty budynków, z których inwestor musi wybrać podzbiór spełniając jego oczekiwania. Każdy budynek reprezentowany jest jako prostokąt o pewnej wysokości h, podstawie od punktu a do punktu b, oraz cenie budowy w (gdzie h, a, b i w to liczby naturalne, przy czym $a < b$). W takim budynku może mieszkać $h \cdot (b - a)$ studentów.

Proszę zaimplementować funkcję:

```
def select_buildings(T, p):
    ...
```

która przyjmuje:

- Tablicę `T` zawierająca opisy `n` budynków. Każdy opis to krotka postaci `(h, a, b, w)`, zgodnie z oznaczeniami wprowadzonymi powyżej.
- Liczbę naturalną `p` określającą limit łącznej ceny wybudowania budynków.

Funkcja powinna zwrócić tablicę z numerami budynków (zgodnie z kolejnością w `T`, numerowanych od 0), które nie zachodzą na siebie, kosztują łącznie mniej niż `p` i mieszczą maksymalną liczbę studentów. Jeśli więcej niż jeden zbiór budynków spełnia warunki zadania, funkcja powinna zwrócić zbiór o najmniejszym łącznym koszcie budowy. Dwa budynki nie zachodzą na siebie, jeśli nie mają punktu wspólnego.

Można założyć, że zawsze istnieje rozwiązanie zawierające co najmniej jeden budynek. Funkcja powinna być możliwie jak najszybsza i zużywać jak najmniej pomięci. Należy bardzo skrótowo uzasadnić jej poprawność i oszacować złożoność obliczeniową.

Przykład. Dla argumentów:

```
T = [ (2, 1, 5, 3),
      (3, 7, 9, 2),
      (2, 8, 11, 1) ]
p = 5
```

wynikiem jest tablica: `[ 0, 2 ]`

[2pkt.] Zadanie 2.

Szablon rozwiązania: `zad2.py`

Dany jest graf nieskierowany $G = (V, E)$ oraz dwa wierzchołki $s, t \in V$. Proszę zaproponować i zaimplementować algorytm, który sprawdza, czy istnieje taka krawędź $\{p, q\} \in E$, której usunięcie z E spowoduje wydłużenie najkrótszej ścieżki między s a t (usuwamy tylko jedną krawędź). Algorytm powinien być jak najszybszy i używać jak najmniej pamięci. Proszę skrótowo uzasadnić jego poprawność i oszacować złożoność obliczeniową.

Algorytm należy zaimplementować jako funkcję:

```
def enlarge(G, s, t):
   ...
```

która przyjmuje graf `G` oraz numery wierzchołków `s`, `t` i zwraca dowolną krawędź spełniającą warunki zadania, lub `None` jeśli takiej krawędzi w `G` nie ma. Graf przekazywany jest jako lista list sąsiadów, t.j. `G[i]` to lista sąsiadów wierzchołka o numerze `i`. Wierzchołki numerowane są od 0. Funkcja powinna zwrócić krotkę zawierającą numery dwóch wierzchołków pomiędzy którymi jest krawędź spełniająca warunki zadania, lub `None` jeśli takiej krawędzi nie ma. Jeśli w grafie oryginalnie nie było ścieżki z `s` do `t` to funkcja powinna zwrócić `None`.

Przykład. Dla argumentów:

```
G = [ [1, 2],
      [0, 2],
      [0, 1] ]
s = 0
t = 2
```

wynikiem jest np. krotka: `(0, 2)`

[2pkt.] Zadanie 3.

Szablon rozwiązania: `zad3.py`

W roku 2050 Maksymilian odbywa podróż przez pustynię z miasta A do miasta B. Droga pomiędzy miastami jest linią prostą na której w pewnych miejscach znajdują się plamy ropy. Maksymilian porusza się 24 kołową cysterną, która spala 1 litr ropy na 1 kilometr trasy. Cysterna wyposażona jest w pompę pozwalającą zbierać ropę z plam. Aby dojechać z miasta A do miasta B Maksymilian będzie musiał zebrać ropę z niektórych plam (by nie zabrakło paliwa), co każdorazowo wymaga zatrzymania cysterny. Niestety, droga jest niebezpieczna. Maksymilian musi więc tak zaplanować trasę, by zatrzymać się jak najmniej razy. Na szczęście cysterna Maksymiliana jest ogromna - po zatrzymaniu zawsze może zebrać całą ropę z plamy (w cysternie zmieściłaby się cała ropa na trasie).

Zaproponuj i zaimplementuj algorytm wskazujący, w których miejscach trasy Maksymilian powinien się zatrzymać i zebrać ropę. Algorytm powinien być możliwe jak najszybszy i zużywać jak najmniej pamięci. Uzasadnij jego poprawność i oszacuj złożoność obliczeniową.

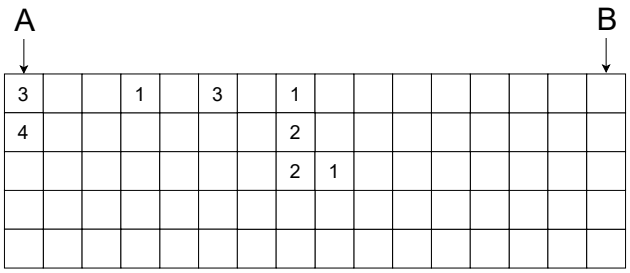
Dane wejściowe reprezentowane są jako dwuwymiarowa tablica liczb naturalnych `T`, w której wartość `T[u][v]` to objętość ropy na polu o współrzędnych (u, v) (objętość 0 oznacza brak ropy). Współrzędne u należą do zbioru $\{0, 1, \ldots, n-1\}$ a współrzędne v to zbioru $\{0, 1, \ldots, m-1\}$. Miasto A znajduje się na polu $(0, 0)$, zaś miasto B na polu $(0, m-1)$. Maksymilian porusza się jedynie po polach $(0, 0)$, $(0, 1), \ldots (0, m-1)$. Bok każdego pola ma długość 1 kilometra. Plamą ropy jest dowolny spójny obszar pól zawierających ropę. Dwa pola należą do spójnego obszaru jeśli mają wspólny bok lub są połączone sekwencją pól (zawierających ropę) o wspólnych bokach. Zakładamy, że początkowo cysterna jest pusta, ale pole $(0, 0)$ jest częścią plamy ropy, którą można zebrać przed wyruszeniem w drogę. Zakładamy również, że zadanie posiada rozwiązanie, t.j. da się dojechać z miasta A do miasta B.

Algorytm należy zaimplementować w funkcji:

```
def plan(T):
   ...
```

która przyjmuje tablicę z opisem zadania i zwraca listę współrzędnych `v` pól na których należy zatrzymać cysternę w celu zebrania ropy (cysterna porusza się po tylko polach $(0, v)$, więc wystarczy zwrócić współrzędną `v`). Lista powinna być posortowana w kolejności postojów. Postój na polu $(0, 0)$ również jest częścią rozwiązania.

Przykład. Dla mapy (w pustych polach są wartości 0, a więc brak ropy):



wynikiem jest np. lista: `[0, 5, 7]`