Algorytmy i Struktury Danych
Offline 5 (5.V.2025)

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. 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, słownik, kolejka `collections.deque`, kolejka priorytetowa (`queue.PriorityQueue, heapq`),
2. korzystanie ze struktur danych dostarczonych razem z zadaniem (jeśli takie są).
3. korzystanie z wbudowanych funkcji sortujących (należy 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: `python3 zad.py`

Szablon rozwiązania:	`zad.py`
Złożoność akceptowalna (2pkt):	$O(n^3)$
Złożoność lepsza (3pkt):	$O(n^2 \log n)$
Złożoność wzorcowa (4pkt):	$O(n^2)$
Gdzie n to liczba zamków.	

Dobrycerz (czyli rycerz, który zawsze uprzejmie mówi "dzień dobry") chce się przedostać z zamku s do zamku t. Mapa zamków dana jest w postaci grafu nieskierowanego G, gdzie każda krawędź ma wagę oznaczającą ile godzin potrzeba, żeby ją przebyć. Wagi to liczby naturalne ze zbioru $\{1, 2, \ldots, 8\}$. Po najdalej 16 godzinach podróży Dobrycerz musi nocować w zamku. Warunki uprzejmości wymagają, żeby spędził w takim zamku 8 godzin (przejazd przez zamki, w których nie nocuje nie kosztuje dodatkowego czasu; szybko mówi "dzień dobry" strażnikom i jedzie dalej). Mapa z której korzysta Dobrycerz ma to do siebie, że liczba dróg jest proporcjonalna do liczby zamków. Czyli jeśli zamków jest n, to wiadomo, że dróg jest $O(n)$.

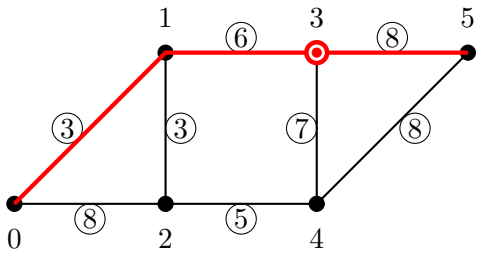
Zadanie polega na implementacji funkcji:

```
goodknight( G, s, t )
```

która na wejściu otrzymuje graf opisujący mapę zamków, reprezentowany w postaci macierzy sąsiedztwa (czyli $G[i][j]$ to liczba godzin, konieczna do przejechania bezpośrednio z zamku i do zamku j; w przypadku braku drogi $G[i][j] = -1$), zamek startowy s oraz zamek docelowy t, i zwraca minimalny czas (wyrażony w godzinach) potrzebny na przejazd z s do t (Dobrycerz nigdy nie musi nocować ani w zamku s ani w zamku t). Można założyć, że zawsze istnieje trasa z zamku s do t.

Przykład. Dla wejścia:

```
#         0  1  2  3  4  5
G = [ [ -1, 3, 8,-1,-1,-1 ],    # 0
      [  3,-1, 3, 6,-1,-1 ],    # 1
      [  8, 3,-1,-1, 5,-1 ],    # 2
      [ -1, 6,-1,-1, 7, 8 ],    # 3
      [ -1,-1, 5, 7,-1, 8 ],    # 4
      [ -1,-1,-1, 8, 8,-1 ]]    # 5
s = 0
t = 5
```



wynikiem jest 25. Dobrycerz pokonuje następującą trasę:

1. Jedzie z zamku 0 do zamku 1 (3 godziny).
2. Jedzie z zamku 1 do zamku 3 (6 godzin).
3. Nocuje w zamku 3 (8 godzin).
4. Jedzie z zamku 3 do zamku 5 (8 godzin)