

CEOI 2026



Dzień 1
(polski)

Day 1
(Polish)

Dziobmistrz

Limit czasu: 10 s Limit pamięci: 512 MiB

Bajtocka Liga Dzięciołów Algorytmicznych ma osobiwie rozbudowaną i ciągle zmieniającą się strukturę wewnętrzną. Liga składa się z n oddziałów, a każdy członek Ligi należy do dokładnie jednego oddziału. Oddziały są ponumerowane od 1 do n , a i -ty oddział ma m_i członków. Zatem Liga ma łącznie $M = m_1 + m_2 + \dots + m_n$ członków.

Każdy oddział jest kierowany przez jednego ze swoich członków, który w tej roli nazywany jest *harnasiem* oddziału. Harnasie są numerowani tak samo jak oddziały, więc (dla każdego $i = 1, \dots, n$) harnaś i stoi na czele oddziału i .

Ponadto harnasie są zorganizowani hierarchicznie poprzez system mentoringu: każdy harnaś z wyjątkiem jednego ma *mentora*, którym jest harnaś innego oddziału. Jedynym harnasiem bez mentora jest *Wielki Dzięcioł* Ligi. Jeśli harnaś a jest mentorem harnasia b , mówimy również, że harnaś b jest *uczniem* harnasia a .

Żaden harnaś nie jest, bezpośrednio ani pośrednio, mentorem samego siebie; dlatego podążając od harnasia do jego mentora, następnie do mentora jego mentora itd., zawsze ostatecznie docieramy do Wielkiego Dzięcioła.

Definiujemy *wpływ* harnasia jako sumę liczby członków w jego oddziale oraz wpływów wszystkich jego uczniów (jeśli tacy istnieją). Łatwo zauważyć, że harnasiem o największym wpływie jest Wielki Dzięcioł, którego wpływ jest zawsze równy M . Harnaś jest nazywany *starszym harnasiem*, jeśli jego wpływ jest $\geq M/2$.

Statut Ligi określa, że starszy harnaś o najmniejszym wpływie (spośród wszystkich starszych harnasi) pełni funkcję *Dziobmistrza* Ligi.

Od czasu do czasu jakiś harnaś (inny niż Wielki Dzięcioł) może *zmienić swoją przynależność*, tak że od tej pory staje się uczniem innego mentora niż wcześniej (pod warunkiem, że jego nowy mentor nie jest jednym z jego uczniów, uczniem któregoś z jego uczniów itd.). W wyniku tego może się zdarzyć, że wpływ niektórych harnasi ulegnie zmianie i rola Dziobmistrza przypadnie innemu harnasiowi niż wcześniej.

Zadanie

Napisz program, który wczytuje opis początkowej struktury Ligi oraz sekwencję zmian przynależności. Program powinien wypisać, kto jest Dziobmistrem w początkowej strukturze Ligi, a także po każdej zmianie przynależności.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n oraz q , oddzielone odstępem; n oznacza liczbę oddziałów, a q liczbę zmian przynależności.

Kolejne n wierszy opisuje początkową strukturę Ligi.

i -ty z tych wierszy zawiera dwie liczby całkowite s_i oraz m_i , oddzielone odstępem; s_i jest mentorem harnasia i (tj. harnasia kierującego oddziałem i), natomiast m_i jest liczbą członków oddziału i . Wartość $s_i = 0$ oznacza, że harnaś i jest Wielkim Dzięciołem i nie ma mentora.

Pozostałe q wierszy opisuje zmiany przynależności.

j -ty z tych wierszy zawiera dwie liczby całkowite \hat{x}_j oraz \hat{z}_j , oddzielone odstępem. Znaczenie tych liczb jest następujące.

Niech t_j (dla $j = 0, \dots, q$) oznacza Dziobmistrza po pierwszych j zmianach przynależności (zatem t_0 jest początkowym Dziobmistrzem przed pierwszą zmianą przynależności). Wówczas j -ta zmiana przynależności polega na tym, że harnaś z_j staje się nowym mentorem harnasia x_j , gdzie

$$x_j = 1 + ((t_{j-1} + \hat{x}_j) \bmod n)$$

oraz

$$z_j = 1 + ((t_{j-1} + \hat{z}_j) \bmod n).$$

Celem takiej reprezentacji wartości x_j oraz z_j jest wymuszenie na programie przetwarzania zmian przynależności w kolejności, w jakiej pojawiają się w danych wejściowych.

Zmiany przynależności w danych wejściowych zawsze będą poprawne, tzn. z_j nie będzie równe x_j , ani nie będzie uczniem x_j , uczniem ucznia x_j itd. Możliwe jest jednak, że z_j już było mentorem x_j bezpośrednio przed j -tą zmianą (tak że w rzeczywistości nie się wtedy nie zmienia).

Należy zauważyć, że jeśli program obliczy błędny wynik t_j w pewnym momencie, to również błędnie zdekoduje kolejne dane wejściowe $\hat{x}_{j+1}, \hat{z}_{j+1}$ itd. Może więc zakończyć się werdyktem RTE (błąd wykonania) zamiast WA (błędna odpowiedź), ponieważ błędnie zdekodowane dane wejściowe mogą być niepoprawne (np. może uzyskać z_{j+1} , który jest uczniem x_{j+1}).

Ograniczenia

- $1 \leq n \leq 1\,000\,000$
- $1 \leq q \leq 30\,000$
- $1 \leq m_i$ dla każdego $i = 1, \dots, n$
- $m_1 + m_2 + \dots + m_n \leq 10^9$
- $1 \leq \hat{x}_j \leq n$ oraz $1 \leq \hat{z}_j \leq n$ dla każdego $j = 1, \dots, q$.

Podzadania

- Podzadanie 1 (15 punktów): $n \leq 100$
- Podzadanie 2 (10 punktów): $n \leq 1000$
- Podzadanie 3 (50 punktów): $n \leq 300\,000$
- Podzadanie 4 (25 punktów): Brak dodatkowych ograniczeń.

Wyjście

Twój program powinien wypisać na wyjście dokładnie $q + 1$ wierszy. W i -tym wierszu należy wypisać jedną liczbę t_{i-1} , tzn. Dziobmistrza po pierwszych $i - 1$ zmianach przynależności.

Oczywiście każda wartość t_j musi być liczbą całkowitą z zakresu $1 \leq t_j \leq n$.

Przykład

| Wejście | Wyjście |
|---------|---------|
| 7 2 | 2 |
| 0 1 | 2 |
| 1 3 | 3 |
| 1 3 | |
| 2 3 | |
| 2 1 | |
| 5 2 | |
| 5 1 | |
| 3 7 | |
| 2 7 | |

Komentarz

Początkowo harnaś 2 jest Dziobmistrem (stąd $t_0 = 2$).

Podczas pierwszej zmiany przynależności odczytujemy $\hat{x}_1 = 3$ oraz $\hat{z}_1 = 7$ i obliczamy $x_1 = 1 + ((2 + 3) \bmod 7) = 6$ oraz $z_1 = 1 + ((2 + 7) \bmod 7) = 3$; zatem harnaś 3 zostaje nowym mentorem harnasia 6; harnaś 2 pozostaje nadal Dziobmistrem (stąd $t_1 = 2$).

Podczas drugiej zmiany przynależności odczytujemy $\hat{x}_2 = 2$ oraz $\hat{z}_2 = 7$ i obliczamy $x_2 = 1 + ((2 + 2) \bmod 7) = 5$ oraz $z_2 = 1 + ((2 + 7) \bmod 7) = 3$; zatem harnaś 3 zostaje nowym mentorem harnasia 5, a także staje się nowym Dziobmistrem (stąd $t_2 = 3$).

DFS

Limit czasu: 1 s Limit pamięci: 256 MiB

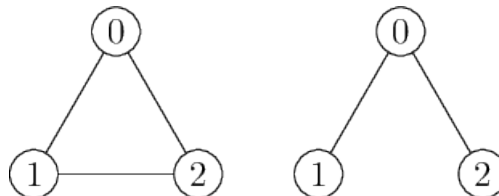
Być może znasz niesławny algorytm DFS (depth-first search, przeszukiwanie w głąb) służący do przeszukiwania grafów. W tym zadaniu będziemy rozważać wyłącznie spójne, nieskierowane grafy proste (bez pętli i multi-krawędzi) z wierzchołkami ponumerowanymi od 0 do $n - 1$, a algorytm DFS będzie wypisywał głębokości i wierzchołki w następujący sposób:

```
DFS(d, v):
    wypisz d/v
    oznacz wierzchołek v jako odwiedzony
    W = lista sąsiadów wierzchołka v uporządkowana rosnąco według numerów
    for each w in W:
        if wierzchołek w nie został jeszcze odwiedzony:
            DFS(d + 1, w)
```

Napisz program, który obliczy liczbę różnych grafów, dla których wywołanie `DFS(0, n - 1)` daje dokładnie taki sam wydruk jak ten podany na wejściu. Na przykład wyjście:

```
0/2
1/0
2/1
```

jest generowane przez wywołanie `DFS(0, 2)` wyłącznie na jednym z dwóch poniższych spójnych nieskierowanych grafów prostych o 3 wierzchołkach:



Wejście

W kolejnych wierszach wejścia znajduje się wynik wywołania `DFS(0, n - 1)` na nieznanym spójnym nieskierowanym grafie prostym o n wierzchołkach. Dane wejściowe składają się zatem z n wierszy w formacie `d/v`, przy czym pierwszy wiersz zawsze ma postać `0/n-1`.

Ograniczenia

- $1 \leq n \leq 2 \cdot 10^5$

Podzadania

- Podzadanie 1 (10 punktów): $n \leq 6$.
- Podzadanie 2 (20 punktów): $n \leq 500$.

- Podzadanie 3 (20 punktów): $n \leq 10^4$.
- Podzadanie 4 (10 punktów): Dla każdego $i \in \{2, \dots, n\}$, i -ty wiersz wejścia ma postać $i-1/i-2$.
- Podzadanie 5 (20 punktów): Dla każdego $i \in \{2, \dots, n\}$, i -ty wiersz wejścia ma postać $i-1/v$ dla pewnego $v \in \{0, \dots, n-2\}$.
- Podzadanie 6 (20 punktów): Brak dodatkowych ograniczeń.

Wyjście

Twój program powinien wypisać na wyjście dokładnie jeden wiersz. W tym wierszu należy wypisać liczbę różnych grafów o żądanej właściwości. Ponieważ liczba ta może być bardzo duża, należy wypisać wynik modulo 1 000 000 007.

Przykład

Wejście

0/2
1/0
2/1

Wyjście

2

Poszukiwanie skarbu

Limit czasu: 8 s Limit pamięci: 256 MiB

Szukasz dawno zaginionego skarbu na ogromnym polu komórek o wymiarach $N \times N$, korzystając z magicznego kompasu. Łącznie na polu ukrytych jest K skrzyń ze skarbami, znajdujących się w różnych komórkach. Twój cel jest prosty: znaleźć je wszystkie!

Kiedy umieścisz kompas na jednej z komórek, znajdzie on najkrótszą ścieżkę do skrzyni ze skarbem, używając wyłącznie ruchów w lewo, do góry, w prawo i w dół (czyli do najbliższej skrzyni według odległości Manhattan). Następnie wskaże kierunek pierwszego kroku. Jeśli istnieje wiele poprawnych pierwszych ruchów zmniejszających dystans do najbliższej skrzyni (lub kilku takich skrzyń), kompas zwróci wszystkie te kierunki. Jeśli dana komórka zawiera skarb, kompas wskaże to zamiast kierunku.

Za każdym razem, gdy znajdziesz skrzynię ze skarbem, możesz opróżnić jej zawartość, ale nie możesz usunąć samej skrzyni — jest zbyt ciężka. Twój kompas nie wie, czy skrzynia jest pełna czy pusta. Będzie wskazywał drogę do najbliższej skrzyni, niezależnie od tego, czy jest pusta, czy pełna.

Spróbuj znaleźć położenie wszystkich skrzyń ze skarbami, używając jak najmniejszej liczby zapytań do kompasu.

Zadanie

To jest zadanie interaktywne. W każdym przypadku testowym (tj. przy każdym uruchomieniu programu) twój program będzie musiał rozwiązać kilka poszukiwań skarbu. Powinieneś komunikować się z systemem sprawdzającym przy użyciu biblioteki dostarczonej przez organizatorów. Biblioteka zawiera następujące deklaracje:

- **void** *NextHunt*(**int** & N , **int** & K) — wywołaj tę funkcję, aby rozpocząć kolejne poszukiwanie skarbu. Zwróci ona rozmiar planszy w parametrze N oraz liczbę skarbów w parametrze K . Jeśli w bieżącym uruchomieniu nie ma już więcej poszukiwań do rozwiązania, funkcja ustawi N i K na -1 ; w takim przypadku powinieneś zakończyć program kodem wyjścia 0 . Zauważ, że możesz wywołać tę funkcję przed odnalezieniem wszystkich skarbów w bieżącym poszukiwaniu, np. jeśli starasz się uzyskać tylko część punktów.
- **enum** { *TREASURE* = 0 , *DIR_RIGHT* = 1 , *DIR_UP* = 2 , *DIR_LEFT* = 4 , *DIR_DOWN* = 8 }; — są to stałe używane przez wartości zwracane przez funkcję *Query* (patrz poniżej).
- **int** *Query*(**int** x , **int** y) — jeśli komórka o współrzędnych (x, y) zawiera skarb, funkcja zwraca *TREASURE*. W przeciwnym razie zwraca sumę jednej lub więcej spośród stałych *DIR_RIGHT*, *DIR_UP*, *DIR_LEFT* i *DIR_DOWN*, wskazując, które ruchy zwraca kompas umieszczony w komórce (x, y) . Współrzędne x i y muszą być liczbami całkowitymi z zakresu od 0 do $N - 1$. (Uwaga: w tym zadaniu współrzędne y rosną od góry do dołu.)

Po tym, jak *NextHunt* zwróci $N = K = -1$, twój program nie może już wywoływać ani *NextHunt*, ani *Query*. Nie może także wywołać *Query* przed pierwszym wywołaniem

NextHunt. Jeśli program nie będzie przestrzegał tych ograniczeń, albo wykona więcej niż 1000 zapytań w trakcie pojedynczego poszukiwania skarbu, albo przekaże do *Query* wartości x i/lub y spoza dozwolonego zakresu, biblioteka zakończy działanie programu i zwróci werdykt run-time-error (RTE) dla bieżącego przypadku testowego.

Skarb uważa się za odnaleziony, jeśli przynajmniej raz wykonano zapytanie dotyczące komórki, która go zawiera. Jeśli program wywoła *NextHunt* przed odnalezieniem wszystkich skarbów w bieżącym poszukiwaniu, nie jest to traktowane jako błąd, ale wpłynie na wynik (więcej informacji o ocenianiu poniżej).

Aby uzyskać dostęp do biblioteki, program powinien dołączyć plik nagłówkowy `treasurehuntlib.h`:

```
#include "treasurehuntlib.h"
```

Plik nagłówkowy można pobrać tutaj: `treasurehuntlib.h`.

Aby pomóc w tworzeniu rozwiązania, dostępna jest prosta implementacja biblioteki: `treasurehuntlib-public.cpp`. Aby skompilować ją wraz z programem, wystarczy dodać jej nazwę jako parametr kompilatora, np.:

```
g++ foo.cpp treasurehuntlib-public.cpp
```

jeśli `foo.cpp` jest nazwą pliku zawierającego twoje rozwiązanie.

Na serwerze oceniającym zostanie użyta inna implementacja biblioteki, więc nie powinieneś zakładać niczego na temat tego, jak dokładnie działa. Możesz jednak założyć, że nie zanieczyszcza globalnej przestrzeni nazw (global namespace) żadnymi deklaracjami poza wymienionymi powyżej (*NextHunt*, *Query* oraz pięcioma stałymi).

Twój kod nie może czytać ze standardowego wejścia ani pisać na standardowe wyjście, ponieważ będą one wykorzystywane przez naszą implementację biblioteki na serwerze oceniającym do komunikacji z resztą środowiska oceny.

Wejście

Ograniczenia

- $1 \leq N \leq 10^6$
- $1 \leq K \leq 3$
- W pojedynczym uruchomieniu programu będzie co najwyżej 100 000 poszukiwań skarbu.
- W pojedynczym poszukiwaniu skarbu można wykonać co najwyżej 1000 zapytań.
- System oceniający nie jest złośliwy.

Podzadania

- Podzadanie 1 (10 punktów) $K = 1$
- Podzadanie 2 (30 punktów) $K = 2$
- Podzadanie 3 (60 punktów) $K = 3$.

Ocenianie

Podzadanie może składać się z wielu przypadków testowych (wielu uruchomień programu), a każdy przypadek testowy może zawierać wiele poszukiwań skarbu. Na potrzeby punktacji wszystkie poszukiwania skarbu z danego podzadania są oceniane wspólnie, niezależnie od tego, jak zostały pierwotnie podzielone pomiędzy przypadki testowe. Dla i -tego poszukiwania skarbu oznaczmy rozmiar planszy przez $N_i \times N_i$, liczbę skarbów przez K_i , liczbę zapytań wykonanych przez program przez Q_i , a liczbę odnalezionych skarbów przez F_i . Oznaczmy ponadto przez S całkowitą liczbę punktów dostępnych za to podzadanie. Wówczas liczba punktów przyznana programowi za to podzadanie wynosi:

- Jeśli program zawsze znajdował wszystkie skarby (tj. jeśli $F_i = K_i$ dla wszystkich i), jego wynik zależy od $t_i = \frac{Q_i}{\lceil \log_2 N_i \rceil}$:

$$\frac{S}{2} + \frac{S}{2} \cdot \min_i f(t_i) \text{ punktów, gdzie } f(t_i) = \begin{cases} 1, & t_i \leq 11 \\ 1 - (t_i - 11)/9, & 11 \leq t_i \leq 20 \\ 0, & t \geq 20. \end{cases}$$

- Jeśli program nie zawsze znajdował wszystkie skarby (tj. istnieje takie i , że $F_i < K_i$), otrzymuje

$$\frac{S}{2} \cdot \min_i \frac{F_i}{K_i} \text{ punktów.}$$

Innymi słowy, połowę punktów otrzymujesz za odnalezienie wszystkich skarbów, a drugą połowę za odnalezienie ich przy użyciu jak najmniejszej liczby zapytań. Aby uzyskać maksymalny wynik, rozwiązanie powinno odnajdywać wszystkie skarby przy użyciu co najwyżej $11 \lceil \log_2 N_i \rceil$ zapytań. Między $11 \lceil \log_2 N_i \rceil$ a $20 \lceil \log_2 N_i \rceil$ zapytaniami wynik maleje liniowo; jeśli natomiast rozwiązanie wykonuje więcej niż $20 \lceil \log_2 N_i \rceil$ zapytań, otrzyma tylko pierwszą połowę punktów za odnalezienie wszystkich skarbów. (Symbole $\lceil \cdot \rceil$ oznaczają zaokrąglenie wartości $\log_2 N_i$ w górę do najbliższej liczby całkowitej.)

Jeśli powyższe wzory dają niecałkowitą liczbę punktów za podzadanie, zostanie ona zaokrąglona do najbliższej liczby całkowitej.

Jeśli program zakończy się błędem wykonania lub nie będzie przestrzegał opisanego wcześniej protokołu korzystania z biblioteki, otrzyma 0 punktów za całe podzadanie. Aby otrzymać częściowe punkty za odnalezienie tylko części skarbów, program musi zatem zakończyć poszukiwanie w poprawny sposób, wywołując *NextHunt*.

Przykład

| Wywołanie | Wartość zwrócona |
|----------------|--------------------------------------|
| NextHunt(N, K) | $N = 4, K = 1$ |
| Query(2, 0) | $DIR_DOWN + DIR_RIGHT = 8 + 1 = 9$ |
| Query(3, 1) | $DIR_DOWN = 8$ |
| Query(3, 2) | $TREASURE = 0$ |
| NextHunt(N, K) | $N = -1, K = -1$ |