Wojciech Rytter
Treść zadania, Opracowanie

Piotr Smulewicz
Program

Dostępna pamięć: 256 MB.

OI, etap III, dzień drugi, 14.04.2016

Klubowicze

Bajtocki Klub Dyskusyjny jest wyjątkowy pod każdym względem. Posiada on 2^n członków, z których każdy zadeklarował, jakie ma poglądy na n fundamentalnych pytań. Konkretne sformułowanie pytań nie jest istotne, wystarczy wiedzieć, że są to pytania, na które można udzielić jednej z dwóch odpowiedzi (np. „kawa czy herbata?"). Poglądy danej osoby możemy kodować za pomocą ciągu bitów, który interpretowany w systemie binarnym da liczbę całkowitą z przedziału od 0 do $2^n - 1$.

W klubie nie ma dwóch osób o jednakowych poglądach. Powiemy, że dwie osoby są **prawie zgodne**, *jeśli ich poglądy różnią się tylko na jednym pytaniu. Ponadto klubowicze to 2^{n-1} panów i 2^{n-1} pań, którzy tworzą 2^{n-1} par. Klubowicze spotykają się przy* **okrągłym** *stole. Chcemy ich tak usadzić, żeby każdy klubowicz siedział obok swojej partnerki lub swojego partnera, a obok siebie po drugiej stronie miał osobę prawie zgodną.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się liczba całkowita n oznaczająca liczbę fundamentalnych pytań. W kolejnych 2^{n-1} wierszach znajdują się opisy par klubowiczów: w i-tym z tych wierszy znajdują się dwie liczby całkowite a_i, b_i ($0 \leqslant a_i, b_i \leqslant 2^n - 1$) oddzielone pojedynczym odstępem, oznaczające, że klubowicze o zestawie poglądów opisanym liczbami a_i i b_i są parą. Każda liczba reprezentująca klubowicza pojawi się na wejściu dokładnie raz.

Wyjście

Jeśli nie istnieje usadzenie klubowiczów spełniające warunki zadania, to w jedynym wierszu standardowego wyjścia należy wypisać jedno słowo `NIE`.

Jeśli takie usadzenie istnieje, to w jedynym wierszu standardowego wyjścia należy wypisać ciąg 2^n liczb całkowitych pooddzielanych pojedynczymi odstępami, oznaczający poprawne usadzenie klubowiczów przy okrągłym stole.

Jeśli istnieje wiele poprawnych odpowiedzi, należy wypisać dowolną z nich.

Przykład

Dla danych wejściowych:

```
3
0 5
4 1
3 6
7 2
```

poprawnym wynikiem jest:

```
0 5 7 2 6 3 1 4
```

Testy „ocen”:

`1ocen`: *$n = 4$, jeśli i jest parzyste, to klubowicze o numerach i oraz $i + 1$ są w parze;*

`2ocen`: *$n = 10$, jeśli i jest nieparzyste, to klubowicze o numerach i oraz $i + 1$ są w parze; wyjątkiem jest klubowicz $2^n - 1$, który jest w parze z klubowiczem 0;*

`3ocen`: *$n = 15$, test losowy, pary na wejściu są posortowane rosnąco względem liczb a_i.*

Ocenianie

Zestaw testów dzieli się na 18 grup, z których każda warta jest 5 albo 6 punktów. W grupie numer k znajdują się wyłącznie testy z $n = k + 1$ (a zatem $2 \leqslant n \leqslant 19$).

Rozwiązanie

Opiszemy rozwiązanie w terminach grafów nieskierowanych, cykli Hamiltona i skojarzeń doskonałych. Dla przypomnienia, cykl Hamiltona to cykl przechodzący przez każdy wierzchołek grafu dokładnie raz. Z kolei skojarzenie w grafie to podzbiór krawędzi, w którym żadne dwie krawędzie nie mają wspólnego końca. Skojarzenie nazywamy *doskonałym*, gdy każdy wierzchołek grafu jest skojarzony (tzn. jest końcem pewnej krawędzi ze skojarzenia).

Rozważmy graf $\mathcal{H}_n$, którego wierzchołki stanowią liczby $0 \leqslant i < 2^n$ traktowane jako binarne ciągi n-elementowe, zaś krawędzie łączą pary liczb różniące się na jednym bicie. Taki graf nazywany jest często *hiperkostką* n-wymiarową jako uogólnienie kwadratu i sześcianu na wyższe wymiary. Natomiast niech $\mathcal{K}_n$ będzie *grafem pełnym* o tym samym zbiorze wierzchołków co $\mathcal{H}_n$, w którym każde dwa różne wierzchołki są połączone krawędzią.

Klubowicze z naszego zadania odpowiadają wierzchołkom grafów. W zadaniu mamy dany zbiór par wierzchołków (skojarzenie doskonałe grafu $\mathcal{K}_n$) i chcemy znaleźć cykl taki, że wierzchołki z każdej pary będą leżały obok siebie na cyklu oraz jeśli wierzchołki leżące obok siebie na cyklu nie są jedną z wejściowych par, to ich numery różnią się dokładnie na jednym bicie (skojarzenie doskonałe grafu $\mathcal{H}_n$).

W terminach grafowych oryginalny problem jest równoważny następującemu:

Wejście: skojarzenie doskonałe X w grafie $\mathcal{K}_n$

Wyjście: cykl Hamiltona $C = X \cup Y$ w grafie $\mathcal{K}_n$ taki, że Y jest skojarzeniem w $\mathcal{H}_n$

Inaczej mówiąc, mając doskonałe skojarzenie w grafie pełnym, chcemy je dopełnić krawędziami z hiperkostki do cyklu Hamiltona. O ile dla grafów niebędących hiperkostką może to być niewykonalne, to w trakcie konstruowania algorytmu przekonamy się, że dzięki specyficznej strukturze grafu $\mathcal{H}_n$ zawsze jesteśmy w stanie znaleźć rozwiązanie.

Głównym pomysłem jest rekurencyjne sprowadzenie problemu do obliczeń na dwóch podhiperkostkach $\mathcal{H}_{n-1}$. Na rozwiązanie składać się będzie: sprytne dołożenie pomocniczych krawędzi, obliczenie rekurencyjnie cykli w podhiperkostkach, następnie

usunięcie pomocniczych krawędzi i dołożenie krawędzi ze zbioru krawędzi łączących podhiperkostki.

Podobnie jak wierzchołki sześcianu można podzielić na dwie grupy znajdujące się na przeciwległych ścianach, tak hiperkostkę $\mathcal{H}_n$ można podzielić na dwa podgrafy izomorficzne z $\mathcal{H}_{n-1}$. W tym celu wystarczy ustalić dowolny indeks i oraz pogrupować ciągi bitowe odpowiadające wierzchołkom $\mathcal{H}_n$ w zależności od wartości i-tego bitu.

Pokażemy, że dla każdych danych wejściowych istnieje rozwiązanie. Jak się nierzadko okazuje, od dowodu niedaleka droga do algorytmu. Zaczniemy od jednej przydatnej definicji. Jeśli C jest cyklem oraz Z jest podzbiorem krawędzi C i jednocześnie skojarzeniem, to przez $Seq(C, Z)$ oznaczmy listę krawędzi Z w kolejności i skierowaniu zadanym przez C (wybieramy dowolne skierowanie cyklu). Dla przykładu jeśli $C = (1, 2, 4, 3, 6, 5, 1)$ oraz $Z = [(5, 6), (2, 4)]$, to $Seq(C, Z) = [(2, 4), (6, 5)]$.

Twierdzenie 1. *Każde skojarzenie doskonałe w grafie $\mathcal{K}_n$ można dopełnić do cyklu Hamiltona skojarzeniem z grafu $\mathcal{H}_n$.*

Dowód: Indukcja po n. Za bazę indukcyjną potraktujemy przypadek dwuwymiarowy. Wtedy graf $\mathcal{H}_n$ jest kwadratem, dla którego teza jest oczywista.

Przypuśćmy, że dla $n \geqslant 3$ dokonaliśmy podziału hiperkostki $\mathcal{H}_n$ na dwie podhiperkostki H_1 i H_2, obie izomorficzne z $\mathcal{H}_{n-1}$. Niech X będzie doskonałym skojarzeniem grafu $\mathcal{K}_n$ i podzielmy krawędzie tego skojarzenia na trzy zbiory:

$$X_1 = X \cap H_1, \qquad X_2 = X \cap H_2, \qquad Z = X - X_1 - X_2.$$

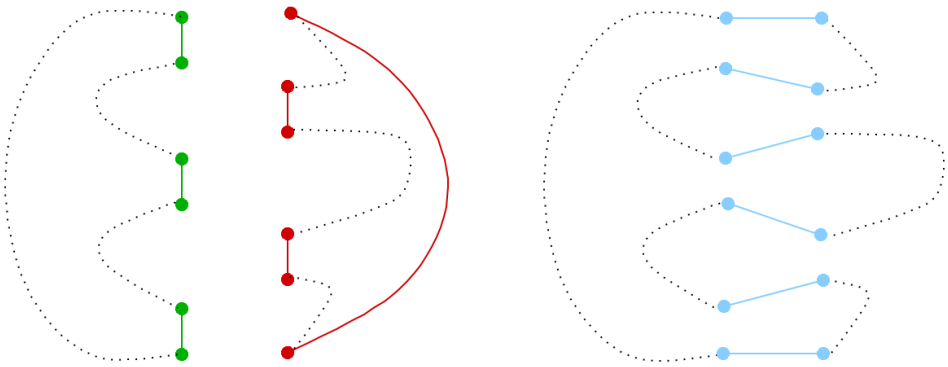
Licząc końce krawędzi w H_1 i H_2, dostajemy $2 \cdot |X_1| + |Z| = 2 \cdot |X_2| + |Z| = 2^{n-1}$, skąd wnioskujemy, że zbiór Z musi zawierać parzyście wiele krawędzi. Są to niejako łączniki pomiędzy podhiperkostkami, które zostaną użyte do połączenia dwóch cykli, które rekurencyjnie skonstruujemy w podhiperkostkach.

Niech dla $i = 1, 2$ zbiór D_i zawiera końce krawędzi ze zbioru Z, które leżą w podhiperkostce H_i. Połączmy elementy D_i w dowolny sposób w pary, uzyskując skojarzenie M_i. Wtedy $X_i \cup M_i$ jest doskonałym skojarzeniem w grafie izomorficznym z $\mathcal{K}_{n-1}$, zatem z założenia indukcyjnego wynika, że uda nam się znaleźć skojarzenie Y_i w H_i, takie że $C_i = X_i \cup M_i \cup Y_i$ będzie cyklem.

Otrzymamy tym samym dwa rozłączne cykle C_1 i C_2 w $\mathcal{H}_n$, które chcielibyśmy połączyć przy pomocy krawędzi z Z. Aby jednak to było możliwe, musimy nieco uważniej skonstruować skojarzenia M_i. Niech $\phi\colon D_1 \to D_2$ będzie przyporządkowaniem końcom krawędzi w Z leżących w H_1 ich końców leżących w H_2. Skojarzenie M_1 możemy wybrać dowolnie, ale skojarzenie M_2 będzie wyznaczone na podstawie cyklu C_1. A konkretnie: dla $Seq(C_1, M_1) = [(v_1, v_2), (v_3, v_4), \ldots, (v_{2k-1}, v_{2k})]$ budujemy cykl $C_2 = X_2 \cup M_2 \cup Y_2$, wybierając

$$M_2 = [(\phi(v_2), \phi(v_3)), (\phi(v_4), \phi(v_5)), \ldots, (\phi(v_{2k}), \phi(v_1))].$$

Teraz możemy już połączyć cykle C_1 i C_2, zastępując zbiór pomocniczych krawędzi $M_1 \cup M_2$ przez krawędzie zbioru Z. Dzięki temu uzyskujemy zbiór $C = X_1 \cup Y_1 \cup X_2 \cup Y_2 \cup Z$. Aby pokazać, że jest on cyklem, wystarczy zauważyć, że powstaje on z C_2 poprzez zastąpienie każdej krawędzi $(\phi(v_i), \phi(v_{i+1})) \in M_2$ przez ścieżkę $\phi(v_i) \to v_i \rightsquigarrow v_{i+1} \to \phi(v_{i+1})$, leżącą poza H_2. Każda taka zamiana jest



Rys. 1: Ilustracja połączenia cykli C_1 i C_2. Po lewej stronie linie ciągłe oznaczają skojarzenia M_1 i M_2. Przerywane linie oznaczają ścieżki leżące wewnątrz podhiperkostek H_1, H_2. Po prawej stronie skojarzenie $M_1 \cup M_2$ zostało zamienione na Z.

niezależna od pozostałych i żadna nie narusza spójności cyklu. Niniejsza konstrukcja została zobrazowana na rysunku 1.

W powyższym rozumowaniu łatwo przeoczyć jeden detal. Przyjęliśmy mianowicie ciche założenie, że zbiór Z jest niepusty. Jest ono istotne, bo w przeciwnym wypadku nie udałoby się nam połączyć obu cykli. Możemy jednak łatwo skonstruować taki podział $\mathcal{H}_n$ na dwie podhiperkostki, w którym to założenie jest spełnione. Wystarczy wybrać dowolną krawędź $e \in X$ i znaleźć bit, na którym jej końce się różnią, a następnie podzielić $\mathcal{H}_n$ w zależności od wartości tego bitu. W ten sposób zagwarantujemy, że co najmniej krawędź e będzie należeć do zbioru Z. ■

Dowód twierdzenia łatwo przetłumaczyć na algorytm rekurencyjny. Poniżej przedstawiamy pseudokod funkcji przyjmującej jako argumenty n, hiperkostkę wymiaru n oraz skojarzenie X, i zwracającej szukany cykl Hamiltona.

```
1:  function FindCycle(n, H, X)
2:  begin
3:    if n = 2 then
4:      return sprawdź oba skojarzenia doskonałe H i wybierz to pasujące do X;
5:    (H_1, H_2) := podział H na podhiperkostki z niepustym łącznikiem;
6:    X_1 := X ∩ H_1,  X_2 := X ∩ H_2,  Z := X − X_1 − X_2;
7:    D_1 := zbiór wierzchołków w H_1 będących końcami krawędzi z Z;
8:    M_1 := dowolne skojarzenie doskonałe w grafie pełnym na zbiorze D_1;
9:    C_1 := FindCycle(n − 1, H_1, X_1 ∪ M_1);
10:   [(v_1, v_2), (v_3, v_4), ..., (v_{2k−1}, v_{2k})] := Seq(C_1, M_1);
11:   M_2 := [(φ(v_2), φ(v_3)), (φ(v_4), φ(v_5)), ..., (φ(v_{2k}), φ(v_1))];
12:   C_2 := FindCycle(n − 1, H_2, X_2 ∪ M_2);
13:   return cykl Hamiltona złożony z (C_1 − M_1) ∪ (C_2 − M_2) ∪ Z;
14: end
```

Dla ustalonego n wszystkie zbiory występujące w funkcji FindCycle mają co najwyżej $N = 2^{n-1}$ elementów, a potrzebne operacje możemy wykonać w czasie liniowym od ich rozmiaru. Złożoność obliczeniową algorytmu można zatem opisać równaniem

$$T(N) = 2 \cdot T(N/2) + O(N),$$

które jest spełnione dla $T(N) = O(N \log N)$. Z rekurencją tego typu można się spotkać na przykład w klasycznym problemie sortowania przez scalanie (ang. *mergesort*). Jest to szczególny przypadek *Twierdzenia o rekurencji uniwersalnej* [6]. Tak więc ostatecznie złożoność rozwiązania to $O(2^{n-1} \log 2^{n-1}) = O(n2^n)$.

Istotną częścią implementacji jest interpretacja rozkładu hiperkostki w języku operacji na maskach bitowych. Kod z komentarzami objaśniającymi poszczególne operacje można znaleźć w pliku `klu2.cpp`.

Wojciech Nadara — Treść zadania, Opracowanie
Bartosz Kostka — Program

Dostępna pamięć: 256 MB. *OI, etap III, dzień drugi, 14.04.2016*

Niebanalne podróże

Bajtazar ostatnio połknął bakcyla podróżowania po Bajtocji. W kraju tym jest n miast (które dla uproszczenia numerujemy liczbami od 1 do n), a bajtocka kolej oferuje podróżnym m dwukierunkowych połączeń kolejowych pomiędzy niektórymi parami miast. Używając tych połączeń, Bajtazar może się dostać do każdego miasta w Bajtocji (być może musi się przy tym przesiadać).

Nasz bohater szczególnie upodobał sobie podróże, w których wyrusza z pewnego miasta, by na końcu do niego wrócić, ale nie odwiedzając przy tym w trakcie podróży żadnego miasta dwukrotnie ani nie używając żadnego połączenia dwukrotnie. Takie podróże nazywa **niebanalnymi**.

W trakcie kolejnej ze swoich wielu podróży Bajtazar zauważył, że każda niebanalna podróż, w którą wyruszył, używała tyle samo połączeń kolejowych. Podejrzewa on, że jest to uniwersalna własność sieci kolejowej w Bajtocji, i poprosił Ciebie o zweryfikowanie tej hipotezy. Ponadto, jeżeli hipoteza jest prawdziwa, to chciałby on także poznać liczbę różnych niebanalnych podróży, które może odbyć. Z wiadomych sobie powodów Bajtazar zadowoli się jedynie resztą z dzielenia liczby takich podróży przez $10^9 + 7$.

Podróż możemy formalnie opisać za pomocą ciągu liczb oznaczających kolejno odwiedzane miasta. Dwie podróże o tej samej długości są różne, jeśli istnieje taki indeks i, że i-te miasta z kolei odwiedzone w trakcie tych podróży są różne. Przez długość podróży rozumiemy liczbę połączeń, których ona używa.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n i m ($n \geqslant 1$, $m \geqslant 0$) oddzielone pojedynczym odstępem, oznaczające odpowiednio liczbę miast w Bajtocji oraz liczbę oferowanych połączeń. Dalej następuje m wierszy opisujących oferowane połączenia. W i-tym z tych wierszy znajdują się dwie liczby całkowite a_i i b_i ($1 \leqslant a_i, b_i \leqslant n$, $a_i \neq b_i$) oddzielone pojedynczym odstępem, oznaczające, że istnieje dwukierunkowe połączenie kolejowe umożliwiające podróż pomiędzy miastami o numerach a_i i b_i. Między każdą parą miast biegnie co najwyżej jedno bezpośrednie połączenie kolejowe.

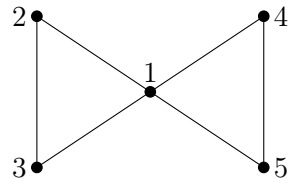
Wyjście

Jeżeli pechowo okazało się, że nie istnieje ani jedna niebanalna podróż, to na standardowe wyjście należy wypisać jedno słowo `BRAK`. *Jeżeli istnieją takie podróże, ale nie wszystkie mają tę samą długość (czyli hipoteza Bajtazara jest fałszywa), należy wypisać jedno słowo* `NIE`. *W końcu jeżeli wszystkie niebanalne podróże mają taką samą długość (czyli hipoteza jest prawdziwa), to należy wypisać jedno słowo* `TAK`, *a w kolejnym wierszu dwie liczby całkowite oddzielone pojedynczym odstępem, oznaczające długość niebanalnych podróży oraz resztę z dzielenia liczby niebanalnych podróży przez $10^9 + 7$.*

Przykład

Dla danych wejściowych:

```
5 6
1 2
2 3
3 1
1 4
4 5
5 1
```



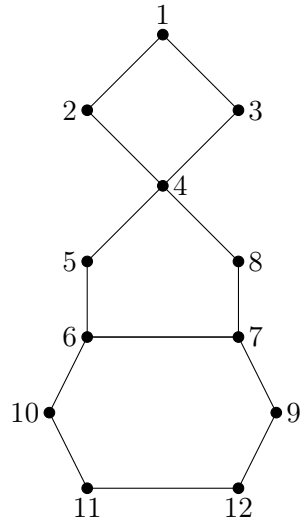
poprawnym wynikiem jest:

```
TAK
3 12
```

Wyjaśnienie do przykładu: *Wszystkie niebanalne podróże mają długość 3 i jest ich 12. Są to kolejno 1-2-3-1, 1-3-2-1, 2-1-3-2, 2-3-1-2, 3-1-2-3, 3-2-1-3, 1-4-5-1, 1-5-4-1, 4-1-5-4, 4-5-1-4, 5-1-4-5, 5-4-1-5.*

Natomiast dla danych wejściowych:

```
12 14
1 2
2 4
3 1
4 3
4 5
5 6
6 7
7 8
8 4
7 9
9 12
12 11
11 10
10 6
```



poprawną odpowiedzią jest:

```
NIE
```

Testy „ocen”:

`1ocen`: *$n = 500\,000$, miasta w Bajtocji leżą na ścieżce, odpowiedź to oczywiście* `BRAK`.

Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n \leqslant 18$	*20*
2	$n, m \leqslant 2000$	*40*
3	$n \leqslant 500\,000$, $m \leqslant 1\,000\,000$	*40*

Rozwiązanie

Pojęcie dwuspójności

Tłumacząc zadanie na język teorii grafów, jesteśmy proszeni o stwierdzenie, czy wszystkie *cykle proste* (tzn. takie, w których żaden wierzchołek ani krawędź się nie powtarzają) w danym grafie nieskierowanym mają taką samą długość. Jeżeli tak jest, to mamy dodatkowo wyznaczyć ich liczbę (i wypisać ją pomnożoną przez dwukrotność tej długości). W tym opracowaniu za każdym razem, gdy będziemy pisali o *cyklach*, będziemy w domyśle mieć na myśli *cykle proste*.

Przypomnijmy pojęcie *grafów dwuspójnych wierzchołkowo*. Są to takie grafy spójne, które po usunięciu dowolnego wierzchołka pozostają spójne. Równoważna definicja stwierdza, że graf dwuspójny to taki graf, w którym dla każdych dwóch krawędzi istnieje cykl prosty je zawierający.

Zbiór krawędzi dowolnego grafu można podzielić na *dwuspójne składowe*. W obrębie dwuspójnej składowej dla każdej pary krawędzi istnieje cykl je zawierający oraz ta własność nie zachodzi dla żadnej pary krawędzi z dwóch różnych składowych. W szczególności, dwuspójną składową może być także pojedyncza krawędź. Żądnym uzasadnienia istnienia takiego podziału zbioru krawędzi można podpowiedzieć, że wystarczy udowodnić, że jeżeli istnieje cykl prosty zawierający krawędzie a oraz b oraz cykl prosty zawierający krawędzie b i c, to istnieje także cykl prosty zawierający krawędzie a i c, co powinno stać się jasne po narysowaniu na kartce kilku przykładów.

Znany jest algorytm działający w złożoności $O(n + m)$ (gdzie n i m to odpowiednio liczby wierzchołków i krawędzi grafu) dzielący krawędzie grafu na dwuspójne składowe. Czytelnikom, którzy o nim nie słyszeli, polecamy się z nim zapoznać np. w książce [4] (jest on także opisany niejawnie w opracowaniu zadania *Blokada* z XV Olimpiady Informatycznej [1]).

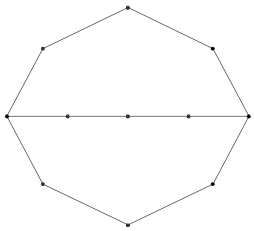
Dlaczego jednak interesujemy się podziałem grafu na dwuspójne dwuspójne? Otóż każdy cykl zawiera się w jednej takiej składowej. Jest to oczywisty wniosek z faktu, że nie istnieje cykl przechodzący przez dwie krawędzie z różnych składowych, co stwierdziliśmy powyżej. Możemy zatem oryginalne zadanie rozwiązać dla każdej dwuspójnej składowej z osobna. Jeżeli w żadnej z nich nie ma żadnego cyklu (czyli każda z nich jest pojedynczą krawędzią, a wejściowy graf jest drzewem), to wypisujemy `BRAK`. Jeżeli w którejkolwiek dwuspójnej składowej istnieją dwa cykle różnych długości, to wypisujemy `NIE`. Załóżmy natomiast, że dla każdej dwuspójnej składowej, w której istnieje cykl, wszystkie cykle są tej samej długości. Jeżeli dla pewnych dwóch dwuspójnych składowych te długości są różne, to wypisujemy `NIE`. Jeżeli jednak dla wszystkich dwuspójnych składowych ta długość jest taka sama, to wszystkie cykle w danym grafie mają tę samą długość i ich liczba jest sumą ich liczności we wszystkich dwuspójnych składowych.

Od tego momentu możemy zatem ograniczyć się do rozwiązania oryginalnego zadania dla grafów dwuspójnych wierzchołkowo, gdyż potrafimy podzielić nasz graf na dwuspójne składowe i potrafimy obliczyć całkowity wynik na podstawie wyników ze wszystkich składowych.

Algorytm dla grafów dwuspójnych

Jeżeli graf dwuspójny ma co najwyżej dwa wierzchołki, to odpowiedzią jest `BRAK`. Spróbujmy znaleźć jakieś przykłady grafów dwuspójnych, w których wszystkie cykle mają tę samą długość. Niewątpliwie takimi grafami są cykle, lecz, jak można się spodziewać, nie są to wszystkie takie grafy.

Wprowadźmy pewien typ grafu, który nazwiemy (c, l)-*cebulą*. Będziemy tak nazywać graf powstały przez połączenie dwóch wierzchołków za pomocą c ścieżek o długości l; patrz rys. 1.



Rys. 1: (3,4)-cebula

Niewątpliwie w (c, l)-cebuli wszystkie cykle są długości $2l$ i jest ich $\binom{c}{2}$. Postawimy teraz odważną hipotezę, na której oprzemy wzorcowy algorytm.

Lemat 1. Cykle oraz (c, l)-cebule (dla $c \geqslant 3$) to jedyne grafy dwuspójne wierzchołkowo, w których wszystkie cykle mają tę samą długość.

Spróbujmy zaprojektować algorytm na podstawie tego lematu. Nad jego prawdziwością zastanowimy się w następnej sekcji.

Naszym celem jest stwierdzenie, czy dany dwuspójny wierzchołkowo graf jest cyklem lub (c, l)-cebulą. Policzmy stopnie wierzchołków naszego grafu (*stopień wierzchołka* to liczba wychodzących z niego krawędzi). Jeżeli wszystkie one są równe dwa, to mamy do czynienia z cyklem (pamiętajmy, że nasz graf jest dwuspójny, zatem w szczególności także spójny). W przeciwnym przypadku, jeżeli nasz graf ma być (c, l)-cebulą, to dokładnie dwa wierzchołki powinny mieć stopień różny od dwóch. Jak jednak stwierdzić, czy graf spełniający taką własność jest (c, l)-cebulą? Nazwijmy te wierzchołki u i v. Weźmy pewną krawędź wychodzącą z u. Jeżeli zaczniemy spacerować od u w kierunku wyznaczonym przez tę krawędź, to napotkamy pewną liczbę wierzchołków o stopniu dwa, aż do momentu, w którym napotkamy albo u albo v. Gdybyśmy napotkali u, to jednak byłoby to sprzeczne z założeniem, że nasz graf jest dwuspójny, zatem napotkany wierzchołek musiał być wierzchołkiem v. Stąd wniosek,

że jeżeli dwuspójny wierzchołkowo graf ma dokładnie dwa wierzchołki o stopniu większym od dwóch, to ma on postać dwóch wierzchołków połączonych zbiorem ścieżek. Pozostaje nam stwierdzić, czy wszystkie owe ścieżki mają taką samą długość. W tym celu wykorzystamy przeszukiwanie grafu wszerz (tzw. BFS) z wierzchołka u, w którym obliczymy odległości wszystkich wierzchołków w grafie od wierzchołka u. Niech $d[w]$ oznacza odległość wierzchołka w od wierzchołka u.

Lemat 2. Graf spełniający wymienione wcześniej warunki jest cebulą wtedy i tylko wtedy, gdy $d[v]$ jest większe od wszystkich pozostałych elementów tablicy d.

Dowód: Łatwo zauważyć, że jeżeli wszystkie ścieżki mają taką samą długość równą l, to wtedy podany warunek jest prawdziwy i $d[v] = l$. Załóżmy przeciwnie, że $d[v] = a$, ale wśród ścieżek łączących u i v pewna ma długość $b > a$. Wtedy wierzchołek poprzedzający v na tej ścieżce znajduje się w odległości od u nie większej niż a (konkretniej, jeżeli nazwiemy go t, to $d[t] = \min(b-1, a+1) \geqslant a = d[v]$), co dowodzi prawdziwości lematu. ■

Jeżeli zatem $d[v]$ jest ściśle największą wartością w tablicy odległości, to nasz graf jest (c, l)-cebulą, przy czym c jest stopniem u, a $l = d[v]$. To kończy opis algorytmu dla grafów dwuspójnych. Pozostaje nam jedynie udowodnić lemat o charakteryzacji grafów dwuspójnych, w których wszystkie cykle mają taką samą długość.

Dowód charakteryzacji interesujących nas grafów

Dowód lematu 1 będzie opierał się na pojęciu *dekompozycji uchowej*. Niech G będzie grafem dwuspójnym wierzchołkowo. Jeżeli G ma co najmniej trzy wierzchołki, to zawiera on cykl. Wyróżnijmy zatem dowolny cykl w G. Twierdzimy, że cały graf można uzyskać w procesie, w którym wielokrotnie do już uzyskanej części grafu (początkowo jest to ów ustalony cykl) doklejamy ścieżkę, której część wspólna z wcześniejszą częścią grafu to jedynie początkowy i końcowy wierzchołek (w szczególności, ścieżka ta może być pojedynczą krawędzią). W przypadku grafów dwuspójnych wierzchołkowo początkowy i końcowy wierzchołek takiej ścieżki muszą być różne, jednak jeżeli pominiemy to ograniczenie, to otrzymamy analogiczną charakteryzację grafów dwuspójnych krawędziowo. Formalnie, twierdzimy, że istnieje taki ciąg grafów $G_1, \ldots, G_k$, że G_1 to dowolny cykl w G, $G_k = G$ oraz dla każdego $i = 1, \ldots, k-1$ zachodzi własność, że $E(G_{i+1}) \setminus E(G_i)$ to zbiór krawędzi tworzących ścieżkę, której część wspólna z G_i jest równa jej początkowi i końcowi (które są różne). Przyjmujemy konwencję, że do kolejnych grafów G_i należą tylko wierzchołki, z których wychodzi co najmniej jedna krawędź. Taki ciąg grafów nazywamy *dekompozycją uchową* grafu G.

Czemu taka dekompozycja istnieje? Załóżmy, że mamy do czynienia z grafem G_i powstałym w wyniku doklejenia pewnej liczby ścieżek. Jeżeli nie jest on jeszcze równy G, to istnieje jakaś krawędź grafu G niezawarta w G_i, która ma co najmniej jeden swój wierzchołek w grafie G_i. Nazwijmy ją e, a jej końce u i v, gdzie $u \in V(G_i)$. Jeżeli $v \in V(G_i)$, to G_{i+1} to graf G_i z dołożoną krawędzią e. Załóżmy zatem, że $v \notin V(G_i)$. Niech f będzie dowolną krawędzią z G_i. Skoro graf G jest dwuspójny, to istnieje cykl zawierający zarówno e jak i f. Idąc kolejnymi krawędziami tego cyklu, zaczynając od v w kierunku przeciwnym do e, musimy kiedyś dojść do jakiegoś wierzchołka z G_i –

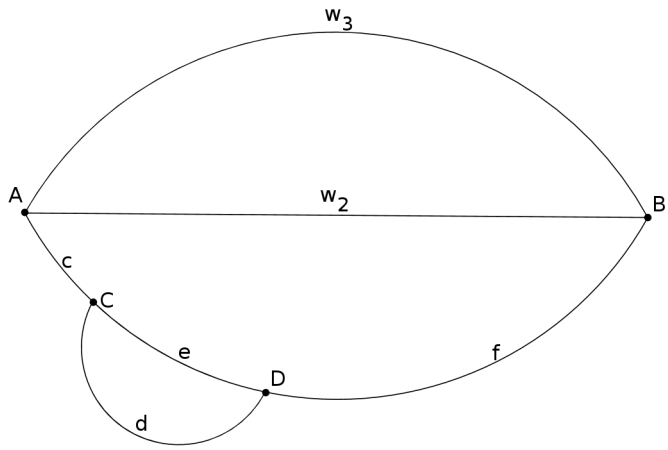
nazwijmy go w. Ten wierzchołek będzie różny od u, gdyż nasz cykl miał być cyklem prostym. Tak stworzona ścieżka – zawierająca e, a potem ścieżkę od v do w – jest ścieżką, którą dokładamy do G_i, aby otrzymać G_{i+1}. Zatem jeżeli G_i nie jest jeszcze równy G, to jesteśmy w stanie dołożyć do niego nowe „ucho". Z racji, że G_{i+1} ma więcej krawędzi niż G_i, ten proces oczywiście musi się kiedyś skończyć.

Zastanówmy się, jak możemy wykorzystać pojęcie dekompozycji uchowej do udowodnienia lematu o charakteryzacji.

Załóżmy, że graf G, w którym zachodzi własność równych długości cykli, sam nie jest cyklem. Wyróżnijmy w nim dowolny cykl G_1. Jesteśmy w stanie do niego dołożyć jakieś ucho. Graf G_2 jest zbiorem trzech ścieżek między dwoma wierzchołkami. Te dwa wierzchołki nazwijmy A i B, a ścieżki nazwijmy w_1, w_2 i w_3. Jeżeli wszystkie cykle w G mają taką samą długość, to ścieżki w_1, w_2 i w_3 mają taką samą długość. Poprzez rozważenie kilku przypadków udowodnimy, że każde kolejne ucho musi być ścieżką łączącą wierzchołki A i B o tej samej długości co trzy ścieżki w_i.

Jeżeli w jest ścieżką, to niech $|w|$ oznacza jej długość (liczoną jako liczba krawędzi). Jeżeli $p_1, \ldots, p_k$ są ścieżkami w grafie, które połączone kolejno tworzą cykl prosty, to ten cykl oznaczmy $p_1p_2 \ldots p_k$, a przez $|p_1p_2 \ldots p_k| = |p_1| + \ldots + |p_k|$ oznaczmy jego długość. Załóżmy, że nowe ucho jest ścieżką pomiędzy C i D. Rozpatrzmy kilka przypadków ze względu na to, gdzie są umiejscowione w grafie te wierzchołki.

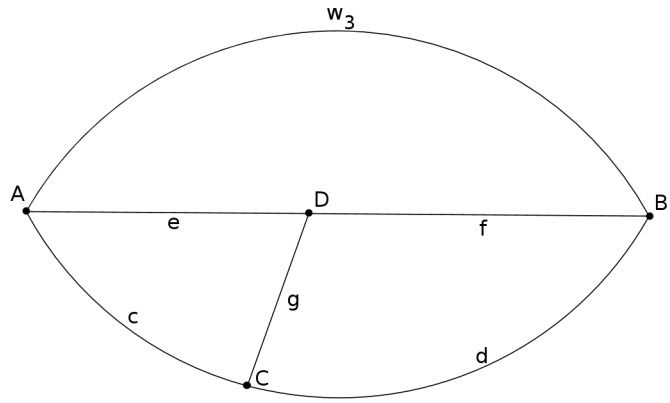
1. C i D oba leżą na jednej ścieżce łączącej A i B (dopuszczamy, że może zachodzić $C \in \{A, B\}$ lub $D \in \{A, B\}$, ale nie oba naraz)



W tym przypadku możemy zauważyć, że cykl de jest krótszy niż cykl dfw_2c, gdyż $|e| < |c| + |e| + |f| = |w_2| < |f| + |w_2| + |c|$. Przeczy to założeniu, że wszystkie cykle proste mają taką samą długość.

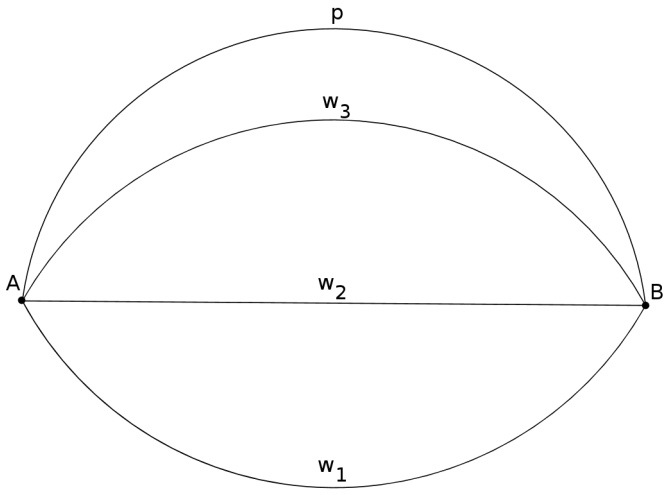
Warto wspomnieć, że na rysunku mogłyby być więcej niż trzy ścieżki łączące A oraz B, jednak istnienie takowych nie jest przeszkodą do uzyskania wspomnianej sprzeczności.

2. C i D leżą we wnętrzach różnych ścieżek łączących A i B



W tym przypadku popatrzmy na cykle $egdw_3$, $cgfw_3$, efw_3 i cdw_3. Mamy $|egdw_3| + |cgfw_3| = (|e| + |f| + |w_3|) + (|c| + |d| + |w_3|) + 2|g| > |efw_3| + |cdw_3|$, skąd wynika, że te cztery cykle nie mogą mieć takich samych długości. Ponownie otrzymujemy sprzeczność.

3. $C = A$ i $B = D$ (lub na odwrót)



Na tym obrazku nowym uchem jest ścieżka p. Jest jasne, że jeżeli wszystkie cykle mają mieć taką samą długość, to musi ona mieć taką samą długość jak każda ze ścieżek w_1, w_2, w_3.

Zaprezentowane przypadki pokrywają wszystkie możliwości. Udowodniliśmy zatem, że każde nowe ucho może być jedynie kolejną ścieżką takiej samej długości łączącą dwa wyróżnione wierzchołki. W ten sposób zakończyliśmy dowód pełnej charakteryzacji dwuspójnych grafów, w których wszystkie cykle są tej samej długości. Rzeczywiście, są to jedynie cykle oraz (c, l)-cebule.

Jacek Tomasiewicz
Treść zadania, Opracowanie

Piotr Smulewicz
Program

Dostępna pamięć: 64 MB.

OI, etap III, dzień drugi, 14.04.2016

Parada

Jak co roku na powitanie wiosny ulicami Bajtogrodu przejdzie Wielka Wiosenna Parada Bajtocka. Swoją obecnością uświetni ją sam Król Bajtazar XVI. Sieć drogowa Bajtogrodu składa się z n skrzyżowań połączonych $n-1$ dwukierunkowymi odcinkami ulic (z każdego skrzyżowania da się dojechać do każdego innego).

Dokładna trasa parady nie jest jeszcze znana, ale wiadomo, że zacznie się ona w jednym ze skrzyżowań, będzie biegła pewną liczbą odcinków ulic i zakończy się na **innym** *skrzyżowaniu. Aby nie zanudzić paradujących, trasa przechodzić będzie przez każdy odcinek ulicy co najwyżej raz.*

Z uwagi na bezpieczeństwo uczestników parady, należy zamknąć bramką wlot każdego odcinka ulicy, przez który nie przechodzi parada, a który wchodzi do skrzyżowania, przez które parada przechodzi (włączając początkowe i końcowe skrzyżowanie). Należy wyznaczyć, ile takich bramek może być potrzebnych.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się liczba całkowita n ($n \geqslant 2$) oznaczająca liczbę skrzyżowań w Bajtogrodzie. Skrzyżowania numerujemy liczbami od 1 do n.

Kolejne $n-1$ wierszy opisuje sieć drogową Bajtogrodu. Każdy z nich zawiera dwie liczby całkowite a i b ($1 \leqslant a,b \leqslant n$, $a \neq b$) oddzielone pojedynczym odstępem, oznaczające, że skrzyżowania o numerach a i b są połączone dwukierunkowym odcinkiem ulicy.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia należy wypisać jedną liczbę całkowitą, oznaczającą maksymalną liczbę bramek, które mogą być potrzebne do zabezpieczenia parady.

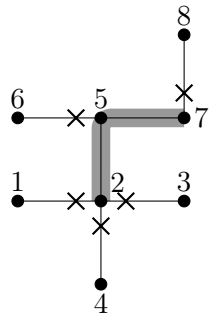
Przykład

Dla danych wejściowych:

```
8
1 2
2 3
4 2
5 2
6 5
5 7
7 8
```

poprawnym wynikiem jest:

```
5
```



Wyjaśnienie do przykładu: *Jeśli parada ruszy ze skrzyżowania 2 i zakończy się na skrzyżowaniu 7, to potrzebne będzie 5 bramek (3 do zamknięcia wlotów do skrzyżowania 2 i po jednej do zamknięcia wlotów do skrzyżowań 5 i 7).*

Testy „ocen":

`1ocen`: *$n = 20$, ścieżka;*

`2ocen`: *$n = 20$, gwiazda;*

`3ocen`: *$n = 1000$, losowy test o następującej własności: i-ty odcinek ulicy (dla $i = 1, \ldots, n-1$) łączy skrzyżowanie numer $i+1$ z jednym ze skrzyżowań o mniejszych numerach.*

Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n \leqslant 20$	*15*
2	$n \leqslant 300$	*16*
3	$n \leqslant 3000$	*22*
4	$n \leqslant 200\,000$	*47*

Rozwiązanie

W zadaniu mamy dane drzewo zawierające n węzłów. Szukamy najtrudniejszej w zabezpieczeniu trasy parady, czyli takiej ścieżki w tym drzewie, z którą połączonych jest bezpośrednio jak najwięcej innych węzłów – liczbę tych węzłów nazwiemy *trudnością* trasy.

Rozwiązanie siłowe $O(n^3)$

Sprawdzamy każdą możliwą ścieżkę i liczymy, z iloma węzłami ona sąsiaduje. Jako że wszystkich ścieżek jest $O(n^2)$, a pojedyncze sprawdzenie zajmuje czas liniowy względem długości ścieżki, złożoność czasowa tego rozwiązania to $O(n^3)$.

Rozwiązanie to zaimplementowane jest w pliku `pars4.cpp`. Za poprawne zaprogramowanie takiego rozwiązania na zawodach można było uzyskać około 30% punktów.

Rozwiązanie wolne $O(n^2)$

Sprawdzamy wszystkie możliwe początki ścieżki, ukorzeniając drzewo w każdym z węzłów. Zakładamy, że korzeń jest początkiem ścieżki, która będzie prowadziła w dół drzewa. Wykonując przeszukiwanie drzewa w głąb (DFS), w czasie stałym aktualizujemy liczbę węzłów sąsiadujących ze ścieżką prowadzącą do aktualnie odwiedzanego

węzła. W ten sposób wszystkie ścieżki o ustalonym początku rozpatrujemy w łącznym czasie $O(n)$. Jako że wszystkich początków jest $O(n)$, to złożoność czasowa tego rozwiązania to $O(n^2)$.

Implementacja takiego rozwiązania znajduje się w pliku `pars1.cpp`. Rozwiązanie tego typu otrzymywało na zawodach około 50% punktów.

Rozwiązanie wzorcowe $O(n)$

Ukorzeniamy drzewo w dowolnym węźle. Następnie, zaczynając od liści i poruszając się w górę drzewa, dla każdego węzła v wyznaczamy dwie wartości:

$h[v]$ – trudność najtrudniejszej trasy parady zaczynającej się w węźle v i prowadzącej w dół poddrzewa węzła v,

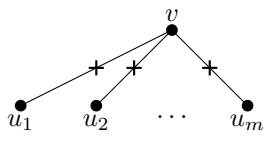
$d[v]$ – trudność najtrudniejszej trasy parady przechodzącej przez v i biegnącej w poddrzewie węzła v.

Jeśli v jest liściem, to oczywiście $h[v] = d[v] = 0$. W ogólnym przypadku, gdy węzeł v ma m synów u_1, u_2, $\ldots$, u_m, dla których obliczyliśmy już wartości $d[u_i]$ i $h[u_i]$, wartości dla węzła v obliczamy z następującej rekurencji:

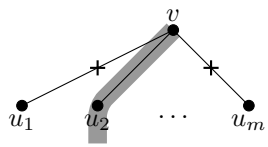
$$h[v] = \max(m, \max_{1 \leqslant i \leqslant m} (h[u_i]) + m - 1)$$

$$d[v] = \max(h[v], \max_{1 \leqslant i < j \leqslant m} (h[u_i] + h[u_j]) + m - 2)$$

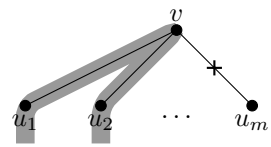
Gdy obliczamy wartość $h[v]$, bierzemy pod uwagę dwie możliwości: albo blokujemy wszystkich synów węzła v (rys. `A`) i trasa parady kończy się w węźle v, albo wybieramy najtrudniejszą trasę parady przechodzącą przez jednego z synów węzła v, blokując przy tym pozostałych synów (rys. `B`). Gdy obliczamy wartość $d[v]$, wybieramy albo trasę parady zaczynającą się w węźle v (co odpowiada wartości $h[v]$), albo połączone dwie trasy parady zaczynające się w synach węzła v (rys. `C`). Na poniższych rysunkach trasę parady zaznaczono kolorem szarym:



rys. `A`



rys. `B`



rys. `C`

Zauważmy, że ostateczny wynik to maksymalna wartość $d[v]$ powiększona o 1, jako że powinniśmy zablokować jeszcze ojca węzła v. Należy tutaj pamiętać o szczególnym przypadku, gdy węzeł v jest korzeniem (wtedy nie dodajemy jedynki).

Wyznaczenie wartości $h[v]$ i $d[v]$ możemy zaimplementować w czasie liniowym od liczby synów m. Faktycznie, wystarczy wyznaczyć maksymalną i drugą co do wielkości wartość $h[u_i]$. Ostatecznie otrzymujemy rozwiązanie działające w czasie liniowym. Przykładową implementację można znaleźć w pliku `par.cpp`.

XXVIII Międzynarodowa Olimpiada Informatyczna,

Kazań, Rosja 2016

Bartosz Kostka, Jakub Radoszewski
Tłumaczenie

Dostępna pamięć: 2 GB, czas: 2 s. *IOI 2016, dzień pierwszy, 14.08.2016*

Kolejka górska

Anna pracuje w parku rozrywki i zajmuje się budową nowej kolejki górskiej. Zaprojektowała już n specjalnych sekcji (dla wygody ponumerowanych od 0 do $n-1$), które zmieniają prędkość wagoników kolejki: są to górki, dolinki i wiele innych. Teraz musi zaproponować ostateczny kształt kolejki, używając wszystkich zaprojektowanych sekcji. W tym zadaniu dla uproszczenia zakładamy, że długość samej kolejki wynosi 0.

Dla każdego i pomiędzy 0 a $n-1$ włącznie, specjalna sekcja numer i ma następujące właściwości:

- *kiedy kolejka wjeżdża do tej sekcji, jej prędkość nie może przekraczać ustalonego limitu: prędkość wagoników musi wynosić* **co najwyżej** s_i *km/h;*
- *kiedy kolejka opuszcza sekcję, jej prędkość wynosi* **dokładnie** t_i *km/h, niezależnie od prędkości, z jaką kolejka wjechała do tej sekcji.*

W ostatecznym projekcie kolejki każdej sekcji należy użyć dokładnie raz. Ponadto pomiędzy każdymi dwiema sąsiadującymi sekcjami można wybudować tory. Anna wybiera kolejność n specjalnych sekcji, a następnie decyduje ona o długości poszczególnych torów. Długość torów jest mierzona w metrach i może być dowolną nieujemną liczbą całkowitą (w szczególności może być to 0). Każdy metr torów pomiędzy specjalnymi sekcjami spowalnia kolejkę o 1 km/h. Na początku trasy kolejka wjeżdża do pierwszej specjalnej sekcji z prędkością 1 km/h.

Ostateczny projekt musi spełniać następujące warunki:

- *w chwili wjeżdżania do specjalnej sekcji kolejka nie może przekraczać limitu prędkości;*
- *w każdym momencie trasy prędkość kolejki musi być dodatnia.*

We wszystkich podzadaniach oprócz trzeciego Twoim zadaniem jest znaleźć kolejność n specjalnych sekcji i dobrać długości torów pomiędzy nimi, tak aby całkowita długość torów była jak najmniejsza. W trzecim podzadaniu musisz jedynie sprawdzić, czy istnieje prawidłowy projekt kolejki, w którym wszystkie tory mają długość 0.

Szczegóły implementacji

Powinieneś zaimplementować następującą funkcję (metodę):

- `int64 plan_roller_coaster(int[] s, int[] t)`
  - `s`: *tablica długości n opisująca maksymalne prędkości wejścia do sekcji.*
  - `t`: *tablica długości n opisująca prędkości wyjścia z sekcji.*
  - *We wszystkich podzadaniach poza trzecim funkcja powinna zwracać minimalną sumaryczną długość wszystkich torów pomiędzy specjalnymi sekcjami. Natomiast w trzecim podzadaniu funkcja powinna zwracać 0, jeżeli istnieje poprawny projekt kolejki, w którym wszystkie tory pomiędzy sekcjami mają długość zero, natomiast dowolną dodatnią liczbę całkowitą, jeżeli taki projekt nie istnieje.*

W języku C sygnatura funkcji jest minimalnie inna:

- `int64 plan_roller_coaster(int n, int[] s, int[] t)`
  - `n`: *rozmiar tablic* `s` *oraz* `t` *(tj. liczba specjalnych sekcji),*
  - *pozostałe parametry są takie same jak powyżej.*

Przykład

```
int64 plan_roller_coaster([1, 4, 5, 6], [7, 3, 8, 6])
```

W tym przykładzie mamy cztery specjalne sekcje. Najlepszym możliwym rozwiązaniem jest wybudowanie ich w kolejności $0, 3, 1, 2$ *i połączenie ich torami o długościach, odpowiednio,* $1, 2, 0$*. Kolejka wtedy porusza się następująco:*

- *Początkowa prędkość kolejki wynosi 1 km/h.*
- *Kolejka rozpoczyna trasę, wjeżdżając do specjalnej sekcji nr 0.*
- *Kolejka opuszcza sekcję nr 0 z prędkością 7 km/h.*
- *Następnie kolejka wjeżdża na tory o długości 1 m. Po ich przejechaniu ma prędkość 6 km/h.*
- *Kolejka wjeżdża do specjalnej sekcji nr 3 z prędkością 6 km/h i opuszcza ją z tą samą prędkością.*
- *Po opuszczeniu sekcji nr 3 kolejka jedzie przez 2 m torów. Prędkość maleje do 4 km/h.*
- *Kolejka wjeżdża do specjalnej sekcji nr 1 z prędkością 4 km/h i opuszcza ją z prędkością 3 km/h.*
- *Natychmiast po opuszczeniu sekcji nr 1 kolejka wjeżdża do specjalnej sekcji nr 2.*
- *Kolejka wyjeżdża z sekcji nr 2. Ostateczna prędkość kolejki wynosi 8 km/h.*

Funkcja powinna zwrócić sumaryczną długość torów pomiędzy specjalnymi sekcjami: $1 + 2 + 0 = 3$*.*

Podzadania

We wszystkich podzadaniach zachodzi $1 \leqslant s_i \leqslant 10^9$ *oraz* $1 \leqslant t_i \leqslant 10^9$*.*

Podzadanie 1 (11 punktów): $2 \leqslant n \leqslant 8$
Podzadanie 2 (23 punkty): $2 \leqslant n \leqslant 16$
Podzadanie 3 (30 punktów): $2 \leqslant n \leqslant 200\,000$*. W tym podzadaniu Twój program musi jedynie sprawdzić, czy wynikiem jest zero, czy też nie. Jeżeli wynikiem nie jest zero, każda dodatnia liczba całkowita jest uznawana za poprawną.*
Podzadanie 4 (36 punktów): $2 \leqslant n \leqslant 200\,000$

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1: liczba całkowita* n,
- *wiersz* $2 + i$, *dla* i *pomiędzy* 0 *i* $n - 1$*: liczby całkowite* s_i *i* t_i.

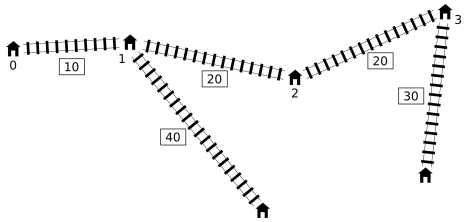
Bartosz Kostka, Jakub Radoszewski
Tłumaczenie

Dostępna pamięć: 2 GB, czas: 2 s. *IOI 2016, dzień pierwszy, 14.08.2016*

Skrót

Paweł ma zabawkową kolejkę elektryczną. Nie jest ona skomplikowana: zawiera ona jedną główną linię torów łączącą n stacji ponumerowanych kolejno od 0 do $n-1$ wzdłuż tej linii. Odległość między stacjami i oraz $i+1$ wynosi l_i centymetrów ($0 \leqslant i < n-1$).

Oprócz głównej linii, w kolejce mogą występować także poboczne linie. Każda poboczna linia łączy stację leżącą na głównej linii z pewną nową stacją, nieleżącą na głównej linii. (Te nowe stacje nie są numerowane). Z każdej stacji głównej linii może wychodzić co najwyżej jedna poboczna linia. Długość pobocznej linii, która zaczyna się na stacji i, wynosi d_i centymetrów. Jeżeli na stacji i nie zaczyna się żadna poboczna linia, to przyjmujemy, że $d_i = 0$.



Paweł chce dodać jeden skrót: ekspresową linię łączącą dwie różne (być może sąsiadujące ze sobą) stacje **głównej linii**. *Ta ekspresowa linia będzie miała długość dokładnie c centymetrów, niezależnie od tego, które dwie stacje połączy.*

Każdy odcinek torów, łącznie z nowo utworzoną ekspresową linią, kolejka może pokonywać w obie strony. **Odległość** *pomiędzy dwiema stacjami rozumiemy jako długość najkrótszej drogi, która łączy te stacje poprzez sieć torów. Z kolei* **średnicą** *całej sieci torów jest największa wśród odległości wszystkich par stacji. Innymi słowy, jest to najmniejsza liczba t, taka że odległość pomiędzy każdymi dwiema stacjami wynosi co najwyżej t centymetrów.*

Paweł chce wybudować ekspresową linię tak, aby średnica powstałej sieci torów była minimalna.

Szczegóły implementacji

Powinieneś zaimplementować jedną funkcję:

- `int64 find_shortcut(int n, int[] l, int[] d, int c)`
  - `n`: *liczba stacji na głównej linii,*
  - `l`: *odległości pomiędzy stacjami na głównej linii (tablica rozmiaru $n-1$),*
  - `d`: *długości pobocznych linii (tablica rozmiaru n),*
  - `c`: *długość nowej, ekspresowej linii.*
  - *Funkcja powinna zwracać najmniejszą możliwą średnicę sieci torów po dodaniu ekspresowej linii.*

Szczegóły implementacji w Twoim języku programowania znajdują się w dostarczonych plikach z szablonami.

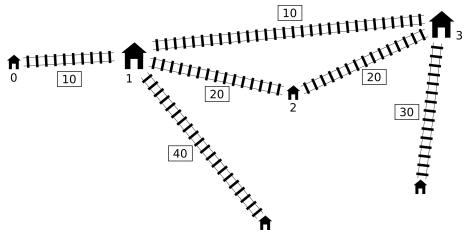
Przykłady

Przykład 1

Dla sieci torów pokazanej powyżej program sprawdzający wykonuje następujące wywołanie:

```
find_shortcut(4, [10, 20, 20], [0, 40, 0, 30], 10)
```

Optymalnym rozwiązaniem jest wybudowanie ekspresowej linii pomiędzy stacjami 1 oraz 3, jak pokazano poniżej.



Średnica nowej sieci torów wynosi 80 centymetrów, zatem funkcja powinna zwrócić wynik 80.

Przykład 2

Program sprawdzający wykonuje następujące wywołanie:

```
find_shortcut(9, [10, 10, 10, 10, 10, 10, 10, 10],
       [20, 0, 30, 0, 0, 40, 0, 40, 0], 30)
```

Optymalne rozwiązanie polega na połączeniu stacji 2 oraz 7. Średnica sieci kolejowej wynosi wtedy 110.

Przykład 3

Program sprawdzający wykonuje następujące wywołanie:

```
find_shortcut(4, [2, 2, 2], [1, 10, 10, 1], 1)
```

Połączenie stacji 1 oraz 2 jest optymalne i zmniejsza średnicę do 21.

Przykład 4

Program sprawdzający wykonuje następujące wywołanie:

```
find_shortcut(3, [1, 1], [1, 1, 1], 3)
```

Połączenie żadnych dwóch stacji ekspresową linią o długości 3 nie poprawi początkowej średnicy sieci torów, wynoszącej 4.

Podzadania

We wszystkich podzadaniach zachodzi $2 \leqslant n \leqslant 1\,000\,000$, $1 \leqslant l_i \leqslant 10^9$, $0 \leqslant d_i \leqslant 10^9$, $1 \leqslant c \leqslant 10^9$.

Podzadanie 1 (9 punktów): $2 \leqslant n \leqslant 10$
Podzadanie 2 (14 punktów): $2 \leqslant n \leqslant 100$
Podzadanie 3 (8 punktów): $2 \leqslant n \leqslant 250$
Podzadanie 4 (7 punktów): $2 \leqslant n \leqslant 500$
Podzadanie 5 (33 punkty): $2 \leqslant n \leqslant 3000$
Podzadanie 6 (22 punkty): $2 \leqslant n \leqslant 100\,000$
Podzadanie 7 (4 punkty): $2 \leqslant n \leqslant 300\,000$
Podzadanie 8 (3 punkty): $2 \leqslant n \leqslant 1\,000\,000$

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1: liczby całkowite* n *i* c,
- *wiersz 2: liczby całkowite* $l_0, l_1, \ldots, l_{n-2}$,
- *wiersz 3: liczby całkowite* $d_0, d_1, \ldots, d_{n-1}$.

Bartosz Kostka, Jakub Radoszewski
Tłumaczenie

Dostępna pamięć: 2 GB, czas: 1 s. *IOI 2016, dzień pierwszy, 14.08.2016*

Wykrywacz cząsteczek

Firma, w której pracuje Petr, skonstruowała maszynę do wykrywania cząsteczek. Każda cząsteczka ma masę wyrażającą się dodatnią liczbą całkowitą. Maszyna ma określony **zakres pomiarowy** *$[l, u]$, przy czym l i u są dodatnimi liczbami całkowitymi. Maszyna może wykryć zbiór cząsteczek wtedy i tylko wtedy, gdy zbiór ten zawiera podzbiór, którego łączna masa należy do zakresu pomiarowego maszyny.*

Formalnie, rozważmy n cząsteczek o masach $w_0, \ldots, w_{n-1}$. Proces wykrywania kończy się powodzeniem, jeśli istnieje zbiór parami różnych indeksów $I = \{i_1, \ldots, i_m\}$ taki że $l \leqslant w_{i_1} + \ldots + w_{i_m} \leqslant u$.

Konstrukcja maszyny gwarantuje, że różnica między u i l jest nie mniejsza niż różnica mas najcięższej i najlżejszej cząsteczki. Formalnie, $u - l \geqslant w_{max} - w_{min}$, gdzie $w_{max} = max(w_0, \ldots, w_{n-1})$ i $w_{min} = min(w_0, \ldots, w_{n-1})$.

Twoim zadaniem jest napisanie programu, który albo wyznaczy jakikolwiek podzbiór zbioru cząsteczek, którego łączna masa należy do zakresu pomiarowego maszyny, albo stwierdzi, że taki podzbiór nie istnieje.

Szczegóły implementacji

Powinieneś napisać jedną funkcję (metodę):

- `int[] solve(int l, int u, int[] w)`
  - `l`, `u`: *końce zakresu pomiarowego,*
  - `w`: *masy cząsteczek.*
  - *Jeśli żądany podzbiór istnieje, funkcja powinna zwrócić tablicę indeksów cząsteczek, które tworzą dowolny taki podzbiór. Jeśli jest więcej niż jedna poprawna odpowiedź, wynikiem funkcji może być dowolna z nich.*
  - *Jeśli żądany podzbiór nie istnieje, funkcja powinna zwrócić pustą tablicę.*

W języku C sygnatura funkcji jest minimalnie inna:

- `int solve(int l, int u, int[] w, int n, int[] result)`
  - `n`: *liczba elementów tablicy* `w` *(tj. liczba cząsteczek),*
  - *pozostałe parametry są takie same jak powyżej.*
  - *Zamiast zwracać tablicę opisującą m indeksów (jak powyżej), funkcja powinna zapisać te indeksy do pierwszych m komórek tablicy* `result` *i zwrócić m.*
  - *Jeśli żądany podzbiór nie istnieje, funkcja nie powinna niczego zapisywać do tablicy* `result` *i powinna zwrócić 0.*

Twój program może zapisać indeksy do zwracanej tablicy (lub do tablicy `result` *w przypadku języka C) w dowolnej kolejności.*

Szczegóły implementacji w Twoim języku programowania znajdują się w dostarczonych plikach z szablonami.

Przykłady

Przykład 1

```
solve(15, 17, [6, 8, 8, 7])
```

W tym przykładzie mamy cztery cząsteczki o masach 6, 8, 8 i 7. Maszyna potrafi wykrywać podzbiory cząsteczek o łącznej masie między 15 a 17 włącznie. Zauważ, że $17 - 15 \geqslant 8 - 6$. *Łączna masa cząsteczek 1 i 3 to* $w_1 + w_3 = 8 + 7 = 15$, *tak więc funkcja może zwrócić* `[1, 3]`. *Inne poprawne odpowiedzi to* `[1, 2]` ($w_1 + w_2 = 8 + 8 = 16$) *i* `[2, 3]` ($w_2 + w_3 = 8 + 7 = 15$).

Przykład 2

```
solve(14, 15, [5, 5, 6, 6])
```

W tym przykładzie mamy cztery cząsteczki o masach 5, 5, 6 i 6 i szukamy podzbioru o łącznej masie między 14 a 15 włącznie. Znów, zauważ że $15 - 14 \geqslant 6 - 5$. *W tym przypadku nie ma żadnego podzbioru cząsteczek o łącznej masie między 14 a 15, więc wynikiem funkcji powinna być pusta tablica.*

Przykład 3

```
solve(10, 20, [15, 17, 16, 18])
```

W tym przykładzie mamy cztery cząsteczki o masach 15, 17, 16 i 18 i szukamy podzbioru o łącznej masie między 10 a 20 włącznie. Znów, zauważ że $20 - 10 \geqslant 18 - 15$. *Każdy podzbiór jednoelementowy ma łączną masę między 10 a 20, tak więc możliwe poprawne wyniki to:* `[0]`, `[1]`, `[2]` *i* `[3]`.

Podzadania

Podzadanie 1 (9 punktów): $1 \leqslant n \leqslant 100$, $1 \leqslant w_i \leqslant 100$, $1 \leqslant u, l \leqslant 1000$, *wszystkie* w_i *są równe.*
Podzadanie 2 (10 punktów): $1 \leqslant n \leqslant 100$, $1 \leqslant w_i, u, l \leqslant 1000$, $max(w_0, \ldots, w_{n-1}) - min(w_0, \ldots, w_{n-1}) \leqslant 1$
Podzadanie 3 (12 punktów): $1 \leqslant n \leqslant 100$, $1 \leqslant w_i, u, l \leqslant 1000$
Podzadanie 4 (15 punktów): $1 \leqslant n \leqslant 10\,000$, $1 \leqslant w_i, u, l \leqslant 10\,000$
Podzadanie 5 (23 punkty): $1 \leqslant n \leqslant 10\,000$, $1 \leqslant w_i, u, l \leqslant 500\,000$
Podzadanie 6 (31 punktów): $1 \leqslant n \leqslant 200\,000$, $1 \leqslant w_i, u, l < 2^{31}$

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1: liczby całkowite* n, l, u.
- *wiersz 2:* n *liczb całkowitych:* $w_0, \ldots, w_{n-1}$.

Bartosz Kostka, Jakub Radoszewski
Tłumaczenie

Dostępna pamięć: 2 GB, czas: 2 s. *IOI 2016, dzień drugi, 16.08.2016*

Obcy

Na jednej z odległych planet nasz satelita wykrył ślady cywilizacji. Na Ziemię dotarło już pierwsze, niskiej rozdzielczości zdjęcie kwadratowego obszaru tej planety. Naszym ekspertom udało się zidentyfikować na zdjęciu n interesujących punktów, gdzie mogą znajdować się ślady życia. Punkty te są ponumerowane od 0 do $n-1$. Kolejnym krokiem będzie wykonanie wysokiej rozdzielczości zdjęć zawierających wszystkie te n punktów.

Na obszarze znajdującym się na zdjęciu (tym o niskiej rozdzielczości) satelita naniósł siatkę o wymiarach m na m złożoną z jednostkowych pól. Tak wiersze, jak i kolumny siatki są ponumerowane kolejno od 0 do $m-1$ (odpowiednio od góry i od lewej). Przez (s,t) oznaczamy pole znajdujące się na przecięciu wiersza s i kolumny t. Interesujący punkt numer i znajduje się na polu (r_i, c_i). Każde pole może zawierać dowolnie wiele interesujących punktów.

Nasz satelita znajduje się na orbicie przebiegającej bezpośrednio nad **główną** *przekątną siatki, przy czym główna przekątna to odcinek łączący lewy górny i prawy dolny róg siatki. Satelita może wykonać wysokiej rozdzielczości zdjęcie dowolnego obszaru, który spełnia następujące warunki:*

- *ma kształt kwadratu,*
- *dwa przeciwległe wierzchołki tego kwadratu znajdują się na głównej przekątnej siatki,*
- *każde pole siatki znajduje się albo całkowicie wewnątrz, albo całkowicie na zewnątrz fotografowanego obszaru.*

Satelita może wykonać co najwyżej k zdjęć w wysokiej rozdzielczości.

Gdy tylko satelita wykona wszystkie zdjęcia, wyśle na Ziemię wysokiej rozdzielczości obraz każdego z fotografowanych pól (niezależnie od tego, czy pole to zawiera jakiekolwiek interesujące punkty). Dane z każdego z tych pól zostaną przesłane dokładnie **raz**, *nawet jeżeli pole to zostało sfotografowane wielokrotnie.*

Tak więc musimy wybrać co najwyżej k kwadratowych obszarów, które zostaną sfotografowane, tak aby:

- *każde z pól zawierających interesujące punkty znalazło się na co najmniej jednym ze zdjęć oraz*
- *liczba pól, które znajdą się na co najmniej jednym zdjęciu, była jak najmniejsza.*

Twoim zadaniem jest wyznaczenie najmniejszej możliwej łącznej liczby pól na zdjęciach.

Szczegóły implementacji

Powinieneś zaimplementować następującą funkcję (metodę):

- `int64 take_photos(int n, int m, int k, int[] r, int[] c)`
  - `n`: *liczba interesujących punktów,*

- `m`: *liczba wierszy (a zarazem liczba kolumn) siatki,*
- `k`: *maksymalna liczba zdjęć, jakie może wykonać satelita,*
- `r`, `c`: *dwie tablice rozmiaru n opisujące współrzędne pól siatki zawierających interesujące punkty. Dla każdego $0 \leqslant i \leqslant n-1$, i-ty z interesujących punktów znajduje się na polu $(r[i], c[i])$.*
- *Funkcja powinna zwrócić najmniejszą możliwą łączną liczbę pól, które znajdą się na co najmniej jednym zdjęciu, przy założeniu, że zdjęcia pokrywają wszystkie spośród interesujących punktów.*

Szczegóły implementacji w Twoim języku programowania znajdują się w dostarczonych plikach z szablonami.

Przykłady

Przykład 1

```
take_photos(5, 7, 2, [0, 4, 4, 4, 4], [3, 4, 6, 5, 6])
```

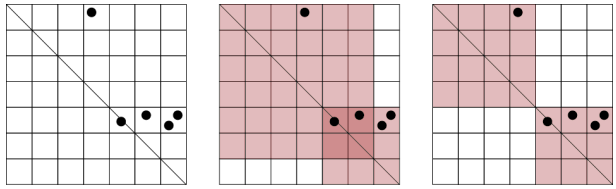
W tym przykładzie mamy siatkę o wymiarach 7×7 zawierającą 5 interesujących punktów. Interesujące punkty znajdują się łącznie na czterech różnych polach: $(0, 3)$, $(4, 4)$, $(4, 5)$ i $(4, 6)$. Satelita może wykonać co najwyżej 2 zdjęcia w wysokiej rozdzielczości.

Jednym ze sposobów uchwycenia wszystkich interesujących punktów jest wykonanie dwóch zdjęć: zdjęcia pokrywającego obszar 6×6 zawierający pola $(0, 0)$ i $(5, 5)$ i zdjęcia pokrywającego obszar 3×3 zawierający pola $(4, 4)$ i $(6, 6)$. Jeśli satelita wykona te dwa zdjęcia, wyśle na Ziemię obrazy 41 pól. Nie jest to optymalny wynik.

Optymalnym rozwiązaniem jest wykonanie jednego zdjęcia pokrywającego obszar 4×4 zawierający pola $(0, 0)$ i $(3, 3)$ i drugiego zdjęcia pokrywającego obszar 3×3 zawierający pola $(4, 4)$ i $(6, 6)$. W ten sposób na zdjęciach znajdzie się łącznie tylko 25 pól, co jest optymalnym wynikiem, więc funkcja `take_photos` *powinna zwrócić 25.*

Zauważ, że pole $(4, 6)$ wystarczy sfotografować raz, mimo iż zawiera ono dwa interesujące punkty.

Przykład ten przedstawiono na poniższych rysunkach. Rysunek po lewej pokazuje siatkę z zaznaczonymi interesującymi punktami. Na środkowym rysunku zaznaczono nieoptymalne rozwiązanie, w którym na zdjęciach znajduje się 41 pól. Rysunek po prawej przedstawia optymalne rozwiązanie.

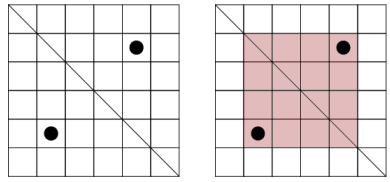


Przykład 2

```
take_photos(2, 6, 2, [1, 4], [4, 1])
```

W tym przypadku mamy 2 interesujące punkty położone symetrycznie, na polach $(1, 4)$ i $(4, 1)$. Każde poprawne zdjęcie zawierające jeden z nich zawiera także drugi z nich. Wystarczy zatem wykonać jedno zdjęcie.

Rysunki poniżej przedstawiają tenże przykład i jego optymalne rozwiązanie. W tym rozwiązaniu satelita wykonuje jedno zdjęcie pokrywające 16 pól.



Podzadania

We wszystkich podzadaniach zachodzi $1 \leqslant k \leqslant n$.

Podzadanie 1 (4 punkty): *$1 \leqslant n \leqslant 50$, $1 \leqslant m \leqslant 100$, $k = n$*
Podzadanie 2 (12 punktów): *$1 \leqslant n \leqslant 500$, $1 \leqslant m \leqslant 1000$, dla każdego i takiego że $0 \leqslant i \leqslant n - 1$, $r_i = c_i$.*
Podzadanie 3 (9 punktów): *$1 \leqslant n \leqslant 500$, $1 \leqslant m \leqslant 1000$*
Podzadanie 4 (16 punktów): *$1 \leqslant n \leqslant 4000$, $1 \leqslant m \leqslant 1\,000\,000$*
Podzadanie 5 (19 punktów): *$1 \leqslant n \leqslant 50\,000$, $1 \leqslant k \leqslant 100$, $1 \leqslant m \leqslant 1\,000\,000$*
Podzadanie 6 (40 punktów): *$1 \leqslant n \leqslant 100\,000$, $1 \leqslant m \leqslant 1\,000\,000$*

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1: liczby całkowite n, m oraz k,*
- *wiersz $2 + i$ $(0 \leqslant i \leqslant n - 1)$: liczby całkowite r_i oraz c_i.*

Bartosz Kostka, Jakub Radoszewski
Tłumaczenie

Dostępna pamięć: 2 GB, czas: 2 s. *IOI 2016, dzień drugi, 16.08.2016*

Obrazek logiczny

Obrazki logiczne to znany typ łamigłówki. W tym zadaniu rozważamy jej prostą, jednowymiarową wersję. Rozwiązujący ma przed sobą rząd n pól. Pola te są ponumerowane od 0 do $n-1$ od lewej do prawej. Zadaniem rozwiązującego jest pokolorować pola na biało i czarno. Czarne pola oznaczamy jako `'X'`*, a białe jako* `'_'`*.*

Rozwiązujący ma zadany ciąg $c = [c_0, \ldots, c_{k-1}]$ złożony z k dodatnich liczb całkowitych, który nazywamy **wskazówką***. Jego zadaniem jest pokolorować pola w taki sposób, aby czarne pola tworzyły dokładnie k bloków kolejnych pól. Ponadto liczba czarnych pól w i-tym bloku od lewej (bloki numerujemy od 0) musi być równa c_i. Przykładowo, jeśli wskazówka to $c = [3, 4]$, to rozwiązanie łamigłówki musi zawierać dwa bloki czarnych pól: jeden o długości 3 i następnie drugi o długości 4. Tak więc jeśli $n = 10$ i $c = [3, 4]$, jednym z rozwiązań spełniających wymagania wskazówki jest* `_XXX__XXXX`*. Zauważmy, że* `XXXX_XXX__` *nie spełnia wymagań wskazówki, jako że bloki czarnych pól znajdują się w złej kolejności. Także* `__XXXXXXX_` *nie spełnia wymagań wskazówki, gdyż zawiera tylko jeden blok czarnych pól, a nie dwa osobne.*

Masz dany częściowo rozwiązany obrazek logiczny, tzn. znasz n i c i wiesz, że niektóre pola muszą być czarne, a niektóre białe. Twoim zadaniem jest wydedukować coś więcej na temat pokolorowania pól.

Mianowicie, przez **poprawne rozwiązanie** *rozumiemy rozwiązanie spełniające wymagania wskazówki, które ponadto jest zgodne z kolorami wskazanych pól. Twój program powinien stwierdzić, które pola w dowolnym poprawnym rozwiązaniu będą pokolorowane na czarno i które pola w dowolnym poprawnym rozwiązaniu będą białe.*

Możesz założyć, że wejście jest dobrane w taki sposób, że istnieje co najmniej jedno poprawne rozwiązanie.

Szczegóły implementacji

Powinieneś napisać jedną funkcję (metodę):

- `string solve_puzzle(string s, int[] c)`
  - `s`: *napis o długości n. Dla każdego i ($0 \leqslant i \leqslant n-1$), znak i to:*
    - `'X'`*, jeśli pole i musi być czarne,*
    - `'_'`*, jeśli pole i musi być białe,*
    - `'.'`*, jeśli nic nie wiadomo o polu i.*
  - `c`: *tablica rozmiaru k zawierająca wskazówkę, zdefiniowana powyżej.*
  - *Funkcja powinna zwrócić napis długości n. Dla każdego i ($0 \leqslant i \leqslant n-1$), znak i wynikowego napisu powinien być równy:*
    - `'X'`*, jeśli pole i jest czarne w każdym poprawnym rozwiązaniu,*
    - `'_'`*, jeśli pole i jest białe w każdym poprawnym rozwiązaniu,*
    - `'?'`*, w przeciwnym przypadku (tzn. jeśli istnieją dwa poprawne rozwiązania, takie że w pierwszym z nich pole i jest czarne, a w drugim białe).*

W języku C sygnatura funkcji jest minimalnie inna:

- `void solve_puzzle(int n, char* s, int k, int* c, char* result)`
  - `n`: *długość napisu* `s` *(liczba pól),*
  - `k`: *rozmiar tablicy* `c` *(długość wskazówki),*
  - *pozostałe parametry są takie same jak powyżej,*
  - *zamiast zwracać napis złożony z n znaków, funkcja powinna go zapisać do napisu* `result`*.*

Kody ASCII znaków występujących w tym zadaniu to:

- *'*`X`*': 88,*
- *'*`_`*': 95,*
- *'*`.`*': 46,*
- *'*`?`*': 63.*

Szczegóły implementacji w Twoim języku programowania znajdują się w dostarczonych plikach z szablonami.

Przykłady

Przykład 1

```
solve_puzzle(".........", [3, 4])
```

Oto wszystkie poprawne rozwiązania łamigłówki:

- `XXX_XXXX__`,
- `XXX__XXXX_`,
- `XXX___XXXX`,
- `_XXX_XXXX_`,
- `_XXX__XXXX`,
- `__XXX_XXXX`.

Można zauważyć, że pola o indeksach (numerowanych od 0) 2, 6 i 7 w każdym poprawnym rozwiązaniu są czarne. Każde inne pole może, ale nie musi być czarne. Poprawną odpowiedzią jest zatem `??X???XX??`*.*

Przykład 2

```
solve_puzzle(".......", [3, 4])
```

W tym przykładzie całe rozwiązanie jest wyznaczone jednoznacznie i poprawną odpowiedzią jest `XXX_XXXX`*.*

Przykład 3

```
solve_puzzle("..._._....", [3])
```

W tym przykładzie możemy wywnioskować, że pole o indeksie 4 musi być białe – nie ma możliwości pokolorowania trzech kolejnych pól na czarno pomiędzy białymi polami o indeksach 3 i 5. Zatem poprawną odpowiedzią jest `???___????`.

Przykład 4

```
solve_puzzle(".X........", [3])
```

Są jedynie dwa poprawne rozwiązania spełniające powyższy opis:

- `XXX_______`,
- `_XXX______`.

Tak więc poprawną odpowiedzią jest `?XX?______`.

Podzadania

We wszystkich podzadaniach zachodzi $1 \leqslant k \leqslant n$ *oraz* $1 \leqslant c_i \leqslant n$ *dla każdego* $0 \leqslant i \leqslant k-1$.

Podzadanie 1 (7 punktów): $n \leqslant 20$, $k = 1$, *s zawiera jedynie* `'.'` *(pusta łamigłówka)*
Podzadanie 2 (3 punkty): $n \leqslant 20$, *s zawiera jedynie* `'.'`
Podzadanie 3 (22 punkty): $n \leqslant 100$, *s zawiera jedynie* `'.'`
Podzadanie 4 (27 punktów): $n \leqslant 100$, *s zawiera jedynie* `'.'` *oraz* `'_'` *(są tylko informacje o białych polach)*
Podzadanie 5 (21 punktów): $n \leqslant 100$
Podzadanie 6 (10 punktów): $n \leqslant 5\,000$, $k \leqslant 100$
Podzadanie 7 (10 punktów): $n \leqslant 200\,000$, $k \leqslant 100$

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1: napis s,*
- *wiersz 2: liczba k, po której następuje k liczb całkowitych* $c_0, \ldots, c_{k-1}$.