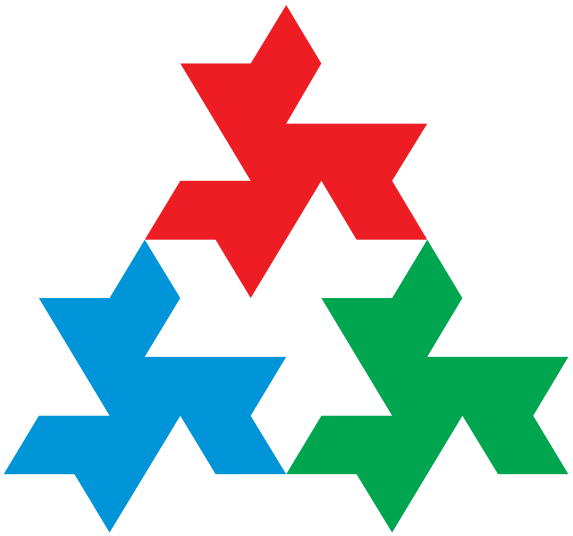
MINISTERSTWO EDUKACJI NARODOWEJ
FUNDACJA ROZWOJU INFORMATYKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ



XIX OLIMPIADA INFORMATYCZNA

2011/2012

OPRACOWANIA ZADAŃ I ETAPU

Olimpiada Informatyczna jest organizowana przy współudziale

WARSZAWA, 2011

MINISTERSTWO EDUKACJI NARODOWEJ
FUNDACJA ROZWOJU INFORMATYKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

XIX OLIMPIADA INFORMATYCZNA
2011/2012
OPRACOWANIA ZADAŃ I ETAPU

WARSZAWA, 2011

Autorzy tekstów:

prof. dr hab. Krzysztof Diks
Marian M. Kędzierski
Alan Kutniewski
Wojciech Śmietanka
Michał Włodarczyk

Autorzy programów:

Igor Adamski
Michał Bejda
Zbigniew Wojna

Redakcja i skład:

mgr Jakub Radoszewski

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Olimpiada Informatyczna jest organizowana przy współudziale

Spis treści

Krzysztof Diks

Wstęp

Drodzy uczestnicy I etapu XIX Olimpiady Informatycznej,

Przekazujemy Wam opisy rozwiązań wzorcowych zadań, z którymi zmierzyliście się podczas I etapu. Prawie 900 uczniów nie przestraszyło się naszej propozycji zadań i wszystkim należą się słowa uznania. Wybierając zadania na I etap, zawsze staramy się, żeby były one interesujące, w zestawie znalazły się propozycje zarówno dla początkujących, jak i zaawansowanych, oraz żeby rozwiązanie każdego zadania sprawiało satysfakcję i czegoś uczyło.

Staraliśmy się docenić wysiłek jak największej liczby osób i dlatego do II etapu zostało zakwalifikowanych ponad 400 uczniów. Macie jeszcze dwa miesiące, żeby przygotować się do II etapu. Zachęcam do intensywnej pracy. Pomoże Wam w tym portal `main.edu.pl`, na którym znajdziecie zadania z wielu konkursów programistycznych, w tym z minionych edycji Olimpiady. Swoje rozwiązania w portalu możecie testować na bieżąco. Korzystajcie też intensywnie z forum Systemu Internetowego Olimpiady, na którym możecie zadawać pytania dotyczące rozwiązań i dzielić się swoimi doświadczeniami w rozwiązywaniu zadań. To jest najlepszy sposób nauki.

Powyższe słowa kieruję też do wszystkich, którzy w tym roku nie zakwalifikowali się do kolejnych etapów Olimpiady. Nie rezygnujcie z dalszej nauki algorytmiki i programowania. To, czego nauczycie się teraz, przyda Wam się nie tylko w konkursach informatycznych, lecz także w przyszłym życiu zawodowym. Umiejętności rozwiązywania problemów algorytmicznych i programowania zdobywane w Olimpiadzie są niezwykle cenione przez najbardziej wymagających i atrakcyjnych pracodawców na rynku informatycznym. Pamiętajcie o tym, że warto uczyć się fundamentów informatyki, do których należą algorytmika i programowanie, a nie tylko narzędzi, które nie dożyją wieku, w którym będziecie podejmować wyzwania zawodowe.

Życzę wszystkim dalszych sukcesów i satysfakcji z wielu godzin przygotowań.

Krzysztof Diks
Przewodniczący Komitetu Głównego Olimpiady Informatycznej

Marian M. Kędzierski
Treść zadania, Opracowanie

Michał Bejda
Program

Dostępna pamięć: 64 MB.

OI, Etap I, 17.10–14.11.2011

Festyn

W Bajtogrodzie trwa festyn charytatywny, a Ty jesteś jedną z kwestujących osób. Ominęło Cię parę atrakcji, w tym bieg z przeszkodami. Bajtazar, miłośnik zagadek logicznych, obiecał przekazać spory datek, jeżeli rozwiążesz jego zagadkę.

Nie znasz dokładnych wyników wyścigu. Bajtazar zdradził Ci jednak część informacji na ich temat, a teraz pyta Cię, ile maksymalnie różnych wyników w wyścigu mogło mieć miejsce. Dwaj uczestnicy mają różne wyniki, jeśli nie przybiegli do mety w tym samym czasie. Dowiedziałeś się, że każdy uczestnik wyścigu uzyskał czas będący całkowitą liczbą sekund. Poznałeś również związki pomiędzy wynikami niektórych par biegaczy. Bajtazar podał Ci część z tych par biegaczy (A, B), dla których wynik A był dokładnie o jedną sekundę lepszy niż wynik B, oraz część z tych par biegaczy (C, D), dla których wynik C był co najmniej tak samo dobry jak wynik D. Napisz program, który pomoże Ci znaleźć rozwiązanie zagadki Bajtazara.

Wejście

Pierwszy wiersz standardowego wejścia zawiera trzy liczby całkowite nieujemne n, m_1, m_2 $(2 \leqslant n \leqslant 600,\ 1 \leqslant m_1 + m_2 \leqslant 100\,000)$, pooddzielane pojedynczymi odstępami, oznaczające odpowiednio: liczbę biegaczy, liczbę poznanych par biegaczy pierwszego typu (A, B) oraz liczbę poznanych par biegaczy drugiego typu (C, D). Biegacze są ponumerowani od 1 do n.

W kolejnych m_1 wierszach podane są pary biegaczy pierwszego typu, po jednej parze w wierszu — i-ty z tych wierszy zawiera dwie liczby a_i i b_i, $a_i \neq b_i$, oddzielone pojedynczym odstępem, oznaczające, że biegacz a_i miał wynik dokładnie o jedną sekundę lepszy niż biegacz b_i.

W kolejnych m_2 wierszach podane są pary biegaczy drugiego typu, po jednej parze w wierszu — i-ty z tych wierszy zawiera dwie liczby c_i i d_i, $c_i \neq d_i$, oddzielone pojedynczym odstępem, oznaczające, że biegacz c_i miał wynik nie gorszy niż biegacz d_i.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia Twój program powinien wypisać jedną liczbę całkowitą równą maksymalnej liczbie różnych wyników czasowych, jakie mogli osiągnąć biegacze przy założeniu podanych kryteriów.

Jeśli nie istnieje kolejność zawodników spełniająca ograniczenia Bajtazara, wyjście powinno składać się z jednego wiersza zawierającego słowo „`NIE`" (bez cudzysłowów).

W testach wartych przynajmniej 15% punktów zachodzi dodatkowe ograniczenie $n \leqslant 10$.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`4 2 2`
`1 2`
`3 4`
`1 4`
`3 1`
	`3`

Wyjaśnienie do przykładu: *Bieg mógł zakończyć się na dwa sposoby:*

1. *pierwszy jest biegacz nr 3, na drugim miejscu są ex aequo biegacze nr 1 i 4, a ostatni jest biegacz nr 2;*
2. *na pierwszym miejscu są ex aequo biegacze nr 1 i 3, a na drugim ex aequo biegacze nr 2 i 4.*

W pierwszej z powyższych możliwości mamy najwięcej różnych wyników czasowych, tzn. trzy.

Rozwiązanie

Analiza problemu

W zadaniu rozważamy n zmiennych całkowitych $t_1, t_2, \ldots, t_n$, oznaczających czasy dotarcia na metę kolejnych uczestników biegu. Pomiędzy niektórymi z tych zmiennych znane są pewne zależności postaci:

1. $t_a + 1 = t_b$
2. $t_c \leqslant t_d$

Przez *wartościowanie* φ zmiennych t_i będziemy rozumieli dowolne przypisanie im wartości całkowitych, które zachowuje podane ograniczenia. *Mocą* wartościowania będziemy nazywali liczbę różnych wartości przypisanych przez to wartościowanie. W zadaniu mamy znaleźć maksymalną moc wartościowania lub stwierdzić, że żadne wartościowanie nie istnieje.

Możemy nieco uprościć sformułowanie zadania przez sprowadzenie dwóch typów zależności do jednego, w którym x_i i y_i oznaczają zmienne, a k_i jest stałą:

$$x_i + k_i \leqslant y_i.$$

Nazwijmy nierówności tej postaci *normalnymi*.

Każdą zależność pierwszego typu możemy zakodować jako dwie nierówności normalne:

$$t_a + 1 \leqslant t_b$$
$$t_b - 1 \leqslant t_a$$

a każdą zależność drugiego typu jako:

$$t_c + 0 \leqslant t_d.$$

Graf nierówności

Rozważmy graf $G = (V, E)$, w którym wierzchołkami są zmienne t_i, zaś krawędzie reprezentują nierówności normalne. Każdej nierówności $x_i + k_i \leqslant y_i$ odpowiada krawędź e_i z wierzchołka x_i do wierzchołka y_i o wadze $w(e_i) = k_i$. Graf ten jest tylko nieco inną reprezentacją układu nierówności normalnych i dalej będziemy traktowali te reprezentacje zamiennie.

Niech $p = (v_1, v_2, \ldots, v_m) \in V^m$ będzie dowolną ścieżką w grafie nierówności. Oznaczmy kolejne krawędzie na tej ścieżce następująco: $e_i = (v_i, v_{i+1})$ dla $i = 1, 2, \ldots, m-1$. Łącząc odpowiadające im nierówności w sekwencję, możemy wywnioskować, że:

$$v_1 \leqslant v_2 - w(e_1) \leqslant v_3 - w(e_1) - w(e_2) \leqslant \ldots \leqslant v_m - w(e_1) - \ldots - w(e_{m-1}), \quad (1)$$

co znaczy, że każde wartościowanie φ musi spełniać zależność $\varphi(v_1) + w(p) \leqslant \varphi(v_m)$, gdzie $w(p) = \sum_{i=1}^{m-1} w(e_i)$ jest wagą ścieżki p.

Widząc to, dla każdej pary wierzchołków $u, v \in V$ od razu chciałoby się znaleźć najdłuższą ścieżkę z u do v, gdyż ona dałaby najsilniejszą zależność spośród wszystkich ścieżek z u do v. Niestety, mogą zdarzyć się takie wierzchołki u, v, dla których nie istnieje żadna ścieżka z u do v. Co więcej, może się zdarzyć, że z pewnego wierzchołka do pewnego innego prowadzą ścieżki dowolnie dużej długości. Prawdopodobnie w większości grafów, które widzieliśmy w życiu, długość ścieżki pomiędzy danymi dwoma wierzchołkami była nieograniczona od góry (zakładając możliwość odwiedzania wierzchołków wielokrotnie). Tutaj sytuacja może być inna, bo w przypadku rozważanego grafu wagi krawędzi często będą ujemne. Niemniej jednak zawsze mogą się w nim zdarzyć ścieżki o nieograniczonej długości.

Aby łatwiej ogarnąć te wszystkie przypadki, zajmijmy się najpierw ostatnim. Co by znaczyło istnienie dowolnie długich ścieżek z pewnego wierzchołka u do pewnego wierzchołka v? Otóż jest to możliwe wtedy i tylko wtedy, gdy graf G zawiera cykl o dodatniej wadze. Jeśli tak jest, to przemierzając taki cykl dowolnie wiele razy, możemy otrzymać dowolnie długą ścieżkę. Z drugiej strony, jeśli graf nie zawierałby cyklu o dodatniej wadze, to z każdej ścieżki można by usunąć wszystkie cykle, nie zmniejszając jej długości, i wtedy otrzymalibyśmy ścieżkę przemierzającą co najwyżej $|V|$ wierzchołków. Jednak ścieżki o ograniczonej liczbie wierzchołków mają też ograniczoną długość, bo wagi pojedynczych krawędzi są ograniczone. Stąd brak dodatniego cyklu oznaczałby, że wszystkie ścieżki w tym grafie mają długości ograniczone z góry.

No dobrze. Co w takim razie zrobić, jeśli graf G zawiera dodatni cykl? Odpowiedź jest bardzo prosta: wypisać „`NIE`”. Weźmy bowiem dowolny taki cykl p i dowolny wierzchołek v na tym cyklu. W podobny sposób, jak wyżej, możemy wywnioskować, że każde wartościowanie φ musi spełnić zależność $\varphi(v) + w(p) \leqslant \varphi(v)$, czyli $w(p) \leqslant 0$, co jest sprzeczne z założeniem o dodatniej długości cyklu. Stąd zaś wniosek, że dla zadanych ograniczeń nie istnieje żadne wartościowanie.

Przyjmijmy teraz, że graf nie zawiera dodatniego cyklu, i dla dowolnych wierzchołków $u, v \in V$ oznaczmy przez $\mathrm{lp}_G(u, v)$ długość najdłuższej ścieżki z wierzchołka u do wierzchołka v w grafie G, pod warunkiem, że istnieje jakakolwiek ścieżka z u do v w G. Wtedy istnieje przynajmniej jedno poprawne wartościowanie. Mówi o tym następujący lemat, który wykorzystamy w dalszej części opisu.

Lemat 1. Układ nierówności normalnych posiada wartościowanie wtedy i tylko wtedy, gdy jego graf nie zawiera cykli o dodatniej wadze.

Dowód: Przed chwilą udowodniliśmy, że istnienie dodatniego cyklu wymusza brak wartościowań, zajmijmy się więc implikacją przeciwną. Załóżmy, że graf G nie zawiera dodatniego cyklu. Pokażemy dla niego wartościowanie.

Najpierw dodajmy do grafu dodatkowy wierzchołek $s \notin V$ oraz krawędzie (s, v) o zerowych wagach dla wszystkich wierzchołków $v \in V$, otrzymując w ten sposób graf G'. Wtedy funkcja $\varphi(v) = \mathrm{lp}_{G'}(s, v)$ jest wartościowaniem dla tak wzbogaconego układu nierówności normalnych. Istotnie, weźmy dowolną nierówność $u + k \leqslant v$ i pokażmy, że jest spełniona. Niech $a = \varphi(u)$, $b = \varphi(v)$. Wtedy mamy $a + k \leqslant b$, ponieważ najdłuższa ścieżka postaci $s \rightsquigarrow u \to v$, o długości $a+k$, jest jedną ze ścieżek z s do v w G', a najdłuższa taka ścieżka ma długość b. A więc rozważana nierówność jest spełniona. Wobec dowolności wyboru nierówności, funkcja φ jest wartościowaniem dla grafu G', a więc także dla G. ■

Silnie spójne składowe grafu nierówności

Weźmy teraz dwa dowolne wierzchołki $u, v \in V$ i zastanówmy się, jakie są możliwe różnice ich wartości $\varphi(v) - \varphi(u)$ dla różnych wartościowań φ. Jeżeli istnieje ścieżka zarówno z u do v, jak i w drugą stronę, to różnica ta jest ograniczona zarówno z dołu, jak i z góry, mamy wtedy bowiem dwie następujące zależności:

$$\varphi(u) + \mathrm{lp}_G(u, v) \leqslant \varphi(v), \quad \varphi(v) + \mathrm{lp}_G(v, u) \leqslant \varphi(u),$$

które można łącznie przedstawić jako:

$$\mathrm{lp}_G(u, v) \leqslant \varphi(v) - \varphi(u) \leqslant -\mathrm{lp}_G(v, u). \tag{2}$$

Do tego wniosku wrócimy później, ale najpierw zastanówmy się, co w przypadku, gdy nie istnieje ścieżka z u do v lub nie istnieje ścieżka z v do u. Mówimy wtedy, że wierzchołki te leżą w różnych *silnie spójnych składowych* grafu G.

Silnie spójna składowa grafu skierowanego to taki podzbiór jego wierzchołków, że pomiędzy każdymi dwoma wierzchołkami z tej składowej istnieją ścieżki w obie strony, zaś dla dowolnego wierzchołka u z tej składowej i wierzchołka v spoza niej nie istnieje ścieżka przynajmniej w jedną stronę pomiędzy nimi. O silnie spójnych składowych można przeczytać w książce [22]. Mają one tę ciekawą własność, że graf silnie spójnych składowych, w którym krawędź ze składowej A prowadzi do składowej B wtedy i tylko wtedy, gdy z każdego (lub równoważnie: dowolnego) wierzchołka $a \in A$ istnieje ścieżka do każdego (lub równoważnie: dowolnego) wierzchołka $b \in B$, jest DAG-iem (ang. *directed acyclic graph*, czyli skierowany graf acykliczny). Przypomnijmy, że każdy DAG można posortować topologicznie, tzn. ustawić jego wierzchołki w ciąg tak, aby każda krawędź prowadziła z lewej strony w prawą.

Korzystając z tych własności, podzielmy graf G na silnie spójne składowe i posortujmy je topologicznie, otrzymując ciąg składowych $V_1, V_2, \ldots, V_s$. Następnie znajdźmy wartościowanie o największej mocy osobno w każdej z nich i oznaczmy te wartościowania odpowiednio przez $\varphi_1, \varphi_2, \ldots, \varphi_s$. Oznaczmy przez K maksymalną

wagę krawędzi w grafie: $K = \max(0, \max_{e \in E} w(e))$, a przez d_i (dla $i = 1, 2, \ldots, s$) najdłuższą ścieżkę w każdej z poszczególnych składowych:

$$d_i = \max_{u,v \in V_i} \mathrm{lp}_{V_i}(u, v).$$

Zauważmy, że dodanie tej samej liczby do wszystkich zmiennych w wartościowaniu φ_i nie wpływa ani na poprawność tego wartościowania, ani na jego moc. Możemy więc tak „przesunąć" każde z wartościowań φ_i, aby dla każdego $i = 1, 2, \ldots, s$ zachodził warunek:

$$\min_{v \in V_i} \varphi_i(v) = (K + 1) \cdot (i - 1) + (d_1 + \ldots + d_{i-1}).$$

Połączmy wartościowania φ_i w funkcję φ na całym zbiorze zmiennych V następująco:

$$\varphi(v) = \varphi_{c(v)}(v),$$

przy czym $c(v)$ oznacza numer składowej, do której należy wierzchołek v. Okazuje się, że tak utworzona funkcja φ jest wartościowaniem dla całego zbioru, którego moc jest równa sumie mocy wartościowań φ_i.

Po pierwsze: dlaczego φ jest wartościowaniem? Oczywiście spełnia ona wszystkie zależności pomiędzy parami wierzchołków leżących w tej samej silnie spójnej składowej. Weźmy więc $u \in V_i$ oraz $v \in V_j$, gdzie $i \neq j$. Bez straty ogólności możemy założyć, że $i < j$. Ponieważ składowe są posortowane topologicznie, to nie może istnieć w grafie składowych krawędź z V_j do V_i, a tym samym nie może istnieć w G krawędź z v do u. Może zaś istnieć krawędź z u do v o pewnej wadze k. Jakie wartości może przyjąć $\varphi(v) - \varphi(u)$? Wiemy, że:

- $\varphi(u) = \varphi_i(u) \in [p_i, p_i + d_i]$, gdzie $p_i = (K + 1) \cdot (i - 1) + (d_1 + \ldots + d_{i-1})$,
- $\varphi(v) = \varphi_j(v) \in [p_j, p_j + d_j]$, gdzie $p_j = (K + 1) \cdot (j - 1) + (d_1 + \ldots + d_{j-1})$.

Tak więc maksymalna wartość $\varphi(u)$ to $p_i + d_i$, a minimalna wartość $\varphi(v)$ to p_j. Stąd:

$$\varphi(v) - \varphi(u) \geqslant p_j - p_i - d_i = (K + 1) \cdot (j - i) + (d_i + \ldots + d_{j-1}) - d_i \geqslant K + 1.$$

Wiemy jednak, że $k < K + 1$, więc nierówność $\varphi(u) + k \leqslant \varphi(v)$ jest spełniona. To pokazuje też, że zbiory wartości poszczególnych wartościowań składowych φ_i są rozłączne, co prowadzi bezpośrednio do wniosku, że moc φ jest równa sumie mocy wszystkich φ_i.

Jaki z tego płynie dla nas wniosek? Bardzo prosty — możemy na początku algorytmu wykonać wstępne obliczenia, polegające na podziale grafu nierówności na silnie spójne składowe, następnie każdą składową zanalizować oddzielnie i zwrócić sumę otrzymanych wyników. Dzielenie grafu na silnie spójne składowe można wykonać przy użyciu dwóch przeszukiwań grafu w głąb (algorytm DFS). Dokładny opis tego algorytmu można znaleźć we wspomnianej już książce [22].

Analiza pojedynczej silnie spójnej składowej

Odtąd możemy już założyć, że rozważany graf nierówności G jest silnie spójny (a więc istnieją w nim ścieżki między każdą parą wierzchołków) oraz nie zawiera dodatnich

cykli. Możemy więc już używać oznaczenia $\mathrm{lp}_G(u,v)$ dla dowolnych wierzchołków $u, v \in V$.

Wróćmy teraz do spostrzeżenia (2). Okazuje się, że podane tam ograniczenie na różnicę wartości $\varphi(v) - \varphi(u)$ jest nie tylko prawdziwe, ale też pokazuje wszystkie możliwe wartości, jakie ta różnica może przyjąć.

Twierdzenie 1. *Niech $G = (V, E)$ będzie silnie spójnym grafem nierówności bez dodatnich cykli i niech $u, v \in V$. Niech Φ będzie zbiorem wszystkich wartościowań zbioru V. Wtedy:*

$$\{\varphi(v) - \varphi(u) : \varphi \in \Phi\} = \mathrm{dint}_G(u,v)$$

gdzie $\mathrm{dint}_G(u,v) = [\mathrm{lp}_G(u,v), -\mathrm{lp}_G(v,u)] \cap \mathbb{Z}$ *(a $\mathbb{Z}$ to zbiór liczb całkowitych).*

Dowód: Wyżej dowiedliśmy już, że $\{\varphi(v) - \varphi(u) : \varphi \in \Phi\} \subseteq \mathrm{dint}_G(u,v)$. Pozostaje pokazać zawieranie odwrotne. Niech k będzie dowolnym elementem przedziału liczb całkowitych $\mathrm{dint}_G(u,v)$. Pokażemy, że istnieje wartościowanie φ, dla którego $\varphi(v) - \varphi(u) = k$.

Dodajmy do grafu G dwie krawędzie odpowiadające dwóm nierównościom normalnym: $u + k \leqslant v$ oraz $v - k \leqslant u$ (razem wyrażającym $v - u = k$); oznaczmy otrzymany graf przez G'. Układ nierówności grafu G posiada wartościowanie φ spełniające $\varphi(v) - \varphi(u) = k$ wtedy i tylko wtedy, gdy G' posiada jakiekolwiek wartościowanie. Z lematu 1 wynika z kolei, że układ nierówności grafu G' posiada wartościowanie wtedy i tylko wtedy, gdy G' nie zawiera dodatniego cyklu. Pozostaje więc jedynie to udowodnić.

Załóżmy przeciwnie, że G' posiada cykl o dodatniej wadze. Weźmy najmniejszy taki cykl C (ze względu na liczbę wierzchołków). Wtedy musi to być cykl prosty (tzn. wierzchołki na nim nie powtarzają się) i musi zawierać przynajmniej jedną z dwóch dodanych krawędzi, ponieważ graf G nie posiadał żadnego dodatniego cyklu. Jeśli C zawiera obie dodane krawędzie, to musi mieć długość 2, ponieważ jest cyklem prostym. Ale suma wag tych dwóch krawędzi wynosi 0, więc C nie jest wtedy cyklem dodatnim.

Załóżmy więc, że na cyklu C leży dokładnie jedna z dodanych krawędzi. Załóżmy, że jest to (u, v) (przeciwny przypadek rozpatruje się symetrycznie). Niech p będzie ścieżką na cyklu C prowadzącą z v do u. Ponieważ cykl C jest prosty, więc ścieżka p musiała istnieć również w oryginalnym grafie G, skąd $w(p) \leqslant \mathrm{lp}_G(v,u)$. Z założenia zaś $k \leqslant -\mathrm{lp}_G(v,u)$. Stąd $w(C) = k + w(p) \leqslant -\mathrm{lp}_G(v,u) + \mathrm{lp}_G(v,u) = 0$, co jest ponownie sprzeczne z założeniem o dodatniości cyklu C.

Tak więc graf G' nie zawiera cyklu o dodatniej wadze, więc istnieje wartościowanie φ układu nierówności odpowiadającego grafowi G spełniające warunek $\varphi(v) - \varphi(u) = k$. ■

Dzięki twierdzeniu 1 jesteśmy już bardzo blisko rozwiązania. Pozostaje nam tylko sformułowanie następującego wniosku:

Wniosek 1. Niech G — graf jak w twierdzeniu 1. Oznaczmy przez D *maksymalną rozciągłość* wartościowania:

$$D = \max_{u,v \in V} \max\{\varphi(v) - \varphi(u) : \varphi \in \Phi\}.$$

Wtedy $D = \max_{u,v \in V}(-\mathrm{lp}_G(u,v))$. Jeżeli ponadto graf G zawiera jedynie krawędzie o wagach -1, 0 oraz 1, to maksymalną mocą wartościowania tego grafu jest $D+1$.

Dowód: Bezpośrednio z twierdzenia 1 wynika, że

$$\max\{\varphi(v) - \varphi(u) : \varphi \in \Phi\} = -\mathrm{lp}_G(v,u).$$

Natomiast drugi sformułowany we wniosku fakt wymaga pewnego komentarza.

Niech φ będzie takim wartościowaniem, a u i v takimi wierzchołkami, że $\varphi(v) - \varphi(u) = D$. Niech $(u_1, u_2, \ldots, u_m)$, $u_1 = u$, $u_m = v$, będzie najdłuższą ścieżką z u do v w G. Ponieważ dla każdego $i = 1, \ldots, m-1$ zachodzi $w(u_i, u_{i+1}) \in \{-1, 0, 1\}$, więc $\varphi(u_{i+1}) - \varphi(u_i) \leqslant 1$, czyli w każdym kroku na tej ścieżce wartościowanie zwiększa się co najwyżej o jeden. Jednak po przejściu całej ścieżki zwiększa się o D. To znaczy, że każda z wartości od $\varphi(u)$ do $\varphi(v)$ musiała być przyjęta przez φ dla pewnej zmiennej u_i. Dostajemy więc $D+1$ różnych wartości zmiennych. Z drugiej strony nie istnieje wartościowanie o większej mocy, bo wtedy jego rozciągłość $\max_{u,v \in V}(\varphi(v) - \varphi(u))$ musiałaby przekroczyć D. ■

Algorytm

Wiemy więc, że dla silnie spójnego grafu nierówności G, nie zawierającego cykli dodatnich, maksymalna moc wartościowania wynosi $1 + \max_{u,v \in V}(-\mathrm{lp}_G(u,v))$. Aby znaleźć tę wartość, wystarczy obliczyć długości najdłuższych ścieżek pomiędzy każdą parą wierzchołków w G. Jak znajdować najdłuższe ścieżki? Wystarczy zmienić znak przy wadze każdej krawędzi, otrzymując graf H, a następnie obliczyć w nim (najkrótsze) odległości $\mathrm{dist}_H(u,v)$ między każdą parą wierzchołków u i v. Ponieważ dla każdej pary wierzchołków u i v zachodzi $\mathrm{dist}_H(u,v) = -\mathrm{lp}_G(u,v)$, więc rozwiązaniem jest liczba $1 + \max_{u,v \in V} \mathrm{dist}_H(u,v)$, czyli średnica grafu H powiększona o jeden.

Jest kilka algorytmów, które pozwalają tę średnicę grafu H obliczyć (pamiętajmy tutaj, że graf H może zawierać krawędzie ujemnej długości):

- n powtórzeń algorytmu Bellmana-Forda, co daje złożoność czasową $O(|V|^2 \cdot |E|)$ (takie rozwiązanie pozwalało uzyskać ok. 50% punktów),
- algorytm Floyda-Warshalla o złożoności czasowej $O(|V|^3)$,
- algorytm Johnsona o złożoności czasowej $O(|V| \cdot |E| + |V|^2 \log |E|)$.

W rozwiązaniach wzorcowych wykorzystano algorytmy Floyda-Warshalla i Johnsona. O wszystkich trzech algorytmach można przeczytać w książce [22].

Możliwe przyspieszenia

Możliwe jest jeszcze dodatkowe zredukowanie grafu, które wprawdzie może przyspieszyć algorytm, ale nie zmniejsza jego asymptotycznej złożoności.

Otóż na samym początku można zidentyfikować wszystkie grupy zmiennych t_i związanych (bezpośrednio lub pośrednio) zależnościami pierwszego typu, tzn.

$t_a + 1 = t_b$. Wtedy wartości zmiennych w jednej takiej grupie są jednoznacznie wyznaczone przez wartość dowolnego jej reprezentanta. Następnie każdą taką grupę można zamienić na pojedynczego reprezentanta lub na dwóch reprezentantów (o największej i najmniejszej wartości w grupie), odpowiednio modyfikując pozostałe nierówności.

To usprawnienie nie było jednak wymagane do uzyskania maksymalnej punktacji.

Testy

Poniższa tabela przedstawia testy wykorzystane do oceny rozwiązań; w szczególności: testy grupy *a* od siódmego włącznie składają się z dużego cyklu i kilku małych składowych sprawdzających poprawność; testy grupy *b* od trzeciego włącznie zostały wygenerowane przez losowe ustawianie zawodników na mecie i ujawnianie losowo zależności pomiędzy nimi; testy grupy *c* to małe testy losowe połączone w DAG; testy grupy *d* to duże testy wydajnościowe wymuszające w niektórych algorytmach dużą liczbę odwiedzin tego samego wierzchołka. Testy z odpowiedzią `NIE` to: *1b*, *4a*, *5a* oraz *6a*.

Liczba *s* podana w opisie testów oznacza liczbę silnie spójnych składowych odpowiedniego grafu nierówności.

Nazwa	n	m_1	m_2	s
fes1a.in	8	3	9	4
fes1b.in	6	3	7	1
fes2a.in	7	2	4	5
fes2b.in	6	2	4	2
fes3a.in	27	12	15	12
fes3b.in	53	71	386	8
fes3c.in	56	40	12	12
fes4a.in	101	20	106	80
fes4b.in	62	41	900	22
fes4c.in	78	60	18	12
fes5a.in	101	20	119	82
fes5b.in	98	17	6 000	49
fes5c.in	250	140	3 129	153
fes6a.in	101	20	144	81
fes6b.in	353	53	6 901	279
fes6c.in	250	155	3 070	159
fes6d.in	600	420	6 365	1
fes7a.in	600	176	397	50
fes7b.in	311	112	3 012	206
fes7c.in	600	718	30 251	290
fes7d.in	600	420	6 365	1
fes8a.in	600	174	390	52
fes8b.in	600	804	5 072	106
fes8c.in	600	763	30 303	283
fes8d.in	600	420	6 365	1
fes9a.in	600	165	404	50
fes9b.in	600	68	50 090	181
fes9c.in	600	795	30 198	279
fes9d.in	600	420	6 365	1
fes10a.in	600	161	416	49
fes10b.in	600	509	50 090	134
fes10c.in	600	816	30 158	281
fes10d.in	600	420	6 365	1
fes11a.in	600	174	390	52
fes11b.in	600	40 139	50 090	30
fes11c.in	600	779	30 198	286
fes11d.in	600	420	6 365	1

Marian M. Kędzierski — Treść zadania, Opracowanie
Zbigniew Wojna — Program

Dostępna pamięć: 128 MB. *OI, Etap I, 17.10–14.11.2011*

Litery

Mały Jaś ma bardzo długie nazwisko. Nie jest jednak jedyną taką osobą w swoim środowisku. Okazało się bowiem, że jedna z jego koleżanek z przedszkola, Małgosia, ma nazwisko dokładnie tej samej długości, chociaż inne. Co więcej, ich nazwiska zawierają dokładnie tyle samo liter każdego rodzaju — tyle samo liter `A`*, tyle samo liter* `B`*, itd.*

Jaś i Małgosia bardzo się polubili i często bawią się razem. Jedną z ich ulubionych zabaw jest zebranie dużej liczby małych karteczek, napisanie na nich kolejnych liter nazwiska Jasia, a następnie przesuwanie karteczek tak, aby powstało z nich nazwisko Małgosi.

Ponieważ Jaś uwielbia łamigłówki, zaczął zastanawiać się, ile co najmniej zamian sąsiednich liter trzeba wykonać, żeby przekształcić jego nazwisko w nazwisko Małgosi. Nie jest to łatwe zadanie dla kilkuletniego dziecka, dlatego Jaś poprosił Ciebie, głównego programistę w przedszkolu, o napisanie programu, który znajdzie odpowiedź na nurtujące go pytanie.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n *(*$2 \leqslant n \leqslant 1\,000\,000$*) oznaczająca liczbę liter w nazwisku Jasia. W drugim wierszu znajduje się* n *kolejnych liter nazwiska Jasia (bez odstępów). W trzecim wierszu znajduje się* n *kolejnych liter nazwiska Małgosi (również bez odstępów). Oba napisy składają się jedynie z wielkich liter alfabetu angielskiego.*

W testach wartych łącznie 30% punktów zachodzi dodatkowy warunek $n \leqslant 1\,000$*.*

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną liczbę całkowitą, oznaczającą minimalną liczbę zamian sąsiednich liter, które przekształcają nazwisko Jasia w nazwisko Małgosi.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`3`
`ABC`
`BCA`
	`2`

Rozwiązanie

Analiza problemu

W zadaniu mamy dane dwa słowa: $j = j_1 j_2 \ldots j_n$ oraz $m = m_1 m_2 \ldots m_n$, w których każda litera pojawia się dokładnie tyle samo razy. Słowa o tej własności będziemy

nazywać *anagramami*. Naszym zadaniem jest znalezienie minimalnej liczby zamian sąsiednich liter, które przekształcają słowo j na słowo m. Dla prostoty takie operacje będziemy nazywali *przesunięciami*.

Powstaje oczywiste pytanie, czy zawsze istnieje sekwencja przesunięć przekształcająca j na m. Nasze założenia mówią, że po posortowaniu liter w słowach j oraz m otrzymujemy to samo słowo. A ponieważ przesunięcia pozwalają na posortowanie dowolnego słowa, możemy skonstruować przykładową strategię w następujący sposób:

- najpierw sortujemy słowo j,
- potem wykonujemy przesunięcia sortujące słowo m, ale w odwrotnej kolejności.

Oczywiście, nie jest to strategia optymalna (np. dla słów „`CBA`" oraz „`BCA`"), jednak jest ona zawsze poprawna.

Dolne ograniczenie

Skoro wiemy już, że przekształcenie j na m przy pomocy przesunięć jest zawsze możliwe, zastanówmy się nad dolnym ograniczeniem na optymalną liczbę przesunięć. W tym celu przyda nam się następujące pojęcie:

Definicja 1. Niech s będzie dowolnym anagramem słowa m. *Dystansem* słowa s od słowa m będziemy nazywać następującą wartość:

$$d(s,m) = \sum_{a \in A} \sum_{i=1}^{k(a)} |s_{a,i} - m_{a,i}|$$

gdzie $A = \{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}\}$ oznacza alfabet, $k(a)$ oznacza liczbę wystąpień litery a w słowie m, zaś $s_{a,1}, \ldots, s_{a,k(a)}$ oraz $m_{a,1}, \ldots, m_{a,k(a)}$ to pozycje kolejnych wystąpień tej litery odpowiednio w słowach s oraz m.

Skąd pomysł na taką funkcję? Otóż intuicyjnie ma ona oddawać sumaryczną odległość pomiędzy „odpowiadającymi sobie"[1] literami w słowach s i m. Dla pokazania dolnego ograniczenia na rozwiązanie zadania nie będzie nam jednak potrzebna wnikliwa analiza funkcji dystansu, a jedynie jej prosta własność:

Fakt 1. *Niech s będzie dowolnym anagramem słowa m oraz niech t będzie słowem powstałym z s przez wykonanie dokładnie jednego przesunięcia. Wtedy* $d(s,m) - d(t,m) \in \{-2, 0, 2\}$.

Dowód: Dowód tego faktu jest bardzo prosty. Każde przesunięcie zmienia o jeden pozycje dwóch liter, przy czym każdą z nich może albo oddalić od „odpowiadającej jej" litery docelowej, albo ją do niej przybliżyć. Stąd dystans może się zmniejszyć o dwa, zwiększyć o dwa lub pozostać taki sam. ■

[1] Zwrot „*odpowiadającymi sobie*" podano w cudzysłowach, ponieważ na tym etapie nie jest jeszcze jasne, czy pierwsza litera `A` w słowie j w optymalnej strategii przechodzi na pierwszą literę `A` w słowie m, druga na drugą, itd. i podobnie dla innych liter.

Prostym wnioskiem z tego faktu jest szukane dolne ograniczenie:

Twierdzenie 1. *Minimalna liczba przesunięć przekształcających słowo j na słowo m jest nie mniejsza niż $\frac{1}{2}d(j,m)$.*

Dowód: Z faktu 1 wynika, że każde przesunięcie zmniejsza dystans co najwyżej o dwa. Tak więc skoro początkowy dystans słowa j od m wynosi $d(j,m)$, a końcowy $d(m,m)=0$, to konieczne jest wykonanie co najmniej $\frac{1}{2}d(j,m)$ przesunięć. ■

Heurystyka polegająca na zwracaniu wyniku $\frac{1}{2}d(j,m)$ jest niepoprawna już dla pary słów „`ABC`" i „`CBA`", ponieważ daje w wyniku 2, podczas gdy poprawnym rozwiązaniem jest 3. Programy implementujące tę heurystykę otrzymywały 0 punktów.

Rozwiązanie poprawne

Tak naprawdę ani powyższe dolne ograniczenie nie daje nam przykładowej strategii (czasem wręcz taka nie istnieje dla rzekomego wyniku $\frac{1}{2}d(j,m)$), ani nawet strategia zaproponowana na początku, bazująca na sortowaniu, nie jest jasna, bo przecież nie widać od razu, jaka jest minimalna liczba przesunięć potrzebnych do posortowania słowa.

Czas więc od rozważań teoretycznych przejść do znalezienia pierwszej konkretnej strategii. Narzucającym się pomysłem jest następujące podejście zachłanne.

Niech $a = m_1 \in A$ będzie pierwszą literą słowa m. Niech $j_{a,1}$ będzie pierwszym wystąpieniem litery a w słowie j. Wykonajmy $j_{a,1}-1$ przesunięć w słowie j przemieszczających pierwsze wystąpienie a na początek słowa, otrzymując nowe słowo j'. Następnie możemy odciąć pierwsze litery ze słów j' oraz m i powtórzyć dokonaną operację n razy.

Czy taka strategia jest optymalna? Często rozwiązania zachłanne są tylko heurystykami, jednak w tym wypadku odpowiedź jest twierdząca. Potrzebny jest jednak na to dowód.

Po pierwsze: dlaczego to właśnie litera z pozycji $j_{a,1}$ w j, a nie jakieś inne wystąpienie tej litery, ma przejść na pierwszą literę słowa m? Załóżmy przeciwnie, że litera odpowiadająca m_1 w strategii optymalnej znajduje się w j na pozycji p, $p > j_{a,1}$. Wtedy w czasie przekształcania j na m musiałby nastąpić moment, w którym litera pochodząca z pozycji $j_{a,1}$ będzie bezpośrednio poprzedzać literę z pozycji p, a kolejne przesunięcie zamieni je ze sobą. Jednak takie przesunięcie niczego nie zmieni, gdyż te dwie litery będą w tym momencie identyczne. Rezygnacja z tego jednego przesunięcia da nam więc strategię lepszą od optymalnej — sprzeczność. Tak więc pierwsza litera słowa m w każdej optymalnej strategii rzeczywiście odpowiada literze z pozycji $j_{a,1}$ w słowie j.

Nazwijmy teraz literę a pochodzącą z pozycji $j_{a,1}$ literą *wyróżnioną*. Rozłóżmy strategię optymalną na dwa niezależnie zmieniające się stany:

- położenie wyróżnionej litery,
- postać *reszty* słowa j, tzn. kolejność wszystkich pozostałych liter.

Dlaczego te stany zmieniają się niezależnie? Zauważmy, że mamy dwa rodzaje przesunięć:

1. zmieniające położenie wyróżnionej litery (wtedy reszta słowa się nie zmienia),
2. zmieniające resztę słowa (wtedy pozycja wyróżnionej litery się nie zmienia).

Niezależność tych dwóch stanów pozwala nam zmienić kolejność wykonywanych przesunięć tak, aby najpierw przeprowadzić wszystkie przesunięcia pierwszego typu, a następnie drugiego. W ten sposób otrzymamy strategię o tej samej liczbie przesunięć, a więc również optymalną. Zarazem pierwsze kroki, przemieszczające literę a w słowie j z pozycji $j_{a,1}$ na początek, będą identyczne jak w zaproponowanym algorytmie zachłannym.

Powtarzając to rozumowanie dla wszystkich kolejnych liter, pokażemy, że litery uznane wyżej za „odpowiadające sobie" rzeczywiście sobie odpowiadają w każdej strategii optymalnej, zaś zaproponowany algorytm zachłanny jest strategią optymalną.

Bezpośrednia implementacja powyższego algorytmu ma złożoność czasową $O(n^2)$. Można ją jednak usprawnić...

Rozwiązanie wzorcowe

Przyspieszenie algorytmu wymaga od nas znalezienia struktury danych, która pozwala szybko wyznaczać pierwszą niewykorzystaną pozycję danej litery w zmienianym słowie, tzn. takie wystąpienie tej litery, które nie zostało jeszcze przemieszczone na początek.

Gdyby słowo j nie ulegało zmianom, łatwo byłoby znajdować pierwsze niewykorzystane pozycje poszczególnych liter. Wystarczyłaby do tego tablica stosów *stosy*[] indeksowana literami alfabetu, która pod indeksem $a \in A$ przechowywałaby pozycje kolejnych wystąpień litery a w słowie j od lewej do prawej. Taką tablicę stosów łatwo zbudować. Wystarczy przejść w pętli po słowie j od końca i każdą napotkaną literę wrzucać na szczyt odpowiadającego jej stosu. Późniejsze korzystanie z tej tablicy również byłoby proste — w każdym kroku algorytmu szukana pozycja litery byłaby na szczycie odpowiadającego jej stosu, a po zdjęciu jej ze szczytu stosu niezmiennik ten byłby zachowany w następnych krokach.

Niestety w naszej sytuacji słowo j zmienia się, potrzebujemy więc jakiegoś mechanizmu, który aktualizowałby pozycje zawarte w tablicy *stosy*[] zgodnie ze zmianami kolejności liter. W każdym momencie możemy skupić się jedynie na niewykorzystanych dotąd literach, bo pozycji liter już przesuniętych na początek nie będziemy później badać. A jak może zmieniać się pozycja niewykorzystanej litery? Otóż w każdym kroku algorytmu będzie ona po prostu większa od początkowej o liczbę tych liter, które na początku znajdowały się dalej niż rozważana, a zostały już wykorzystane.

Musimy zatem zaproponować mechanizm pozwalający szybko znajdować liczbę wykorzystanych liter znajdujących się początkowo w słowie j na pozycji dalszej niż zadana. Do tego służyć może struktura danych zwana *drzewem licznikowym* lub *przedziałowym*. Drzewo licznikowe jest statycznym drzewem binarnym zbudowanym nad tablicą $t[1..n]$ liczb; pozwala ono na wykonywanie w czasie $O(\log n)$ następujących operacji:

$set(i, x)$ — przypisz $t[i] := x$,

$sum(l, r)$ — zwróć sumę $t[l] + t[l + 1] + \ldots + t[r]$, przy czym $l \leqslant r$.

Mając do dyspozycji drzewo licznikowe, możemy już bardzo łatwo utrzymywać informacje o przesunięciach liter w słowie j. Wystarczy na początku, przy budowaniu drzewa licznikowego, wyzerować tablicę $t[]$, a potem, przesuwając literę z jej początkowej pozycji i na początek słowa, wykonywać operację $set(i, 1)$. Wtedy w dowolnym kroku algorytmu wiemy, że pozycja niewykorzystanej litery, która początkowo znajdowała się na pozycji i, od początku algorytmu zwiększyła się o $sum(i+1, n)$. Algorytm wygląda więc następująco:

```
1: wynik := 0;
2: for i := 1 to n do begin
3:     z := m[i];
4:     pos := stosy[z].pop();
5:     set(pos, 1);
6:     pos := pos + sum(pos + 1, n);
7:     wynik := wynik + pos − i;
8: end
9: return wynik;
```

Drzewa przedziałowe pojawiały się już w rozwiązaniach wielu zadań z Olimpiady Informatycznej, nie będziemy więc ich tutaj szczegółowo opisywać. Można o nich przeczytać np. w opracowaniu zadania Tetris 3D z XIII Olimpiady Informatycznej [13].

Nieco inne spojrzenie na zadanie

W powyższym rozwiązaniu wzorcowym korzysta się z drzew przedziałowych, które nie wszystkim są znane. Okazuje się jednak, że nasze zadanie można rozwiązać bez stosowania takich struktur danych.

Pokazaliśmy wcześniej, że *dystans* jest ograniczeniem dolnym na wynik, ale w ogólności wynik może być od niego większy. Istnieje jednak inna miara odległości pomiędzy słowami, którą także można łatwo wyznaczyć, bez symulowania żadnej konkretnej strategii, a która jest już równa szukanemu wynikowi. Aby ją poznać, przyjrzymy się permutacji[2] liter, jaką należy wykonać, by przekształcić słowo j na słowo m. Permutację tę nazwijmy *permutacją przeprowadzającą j na m*.

Wiemy już, że i-ta litera `A` w słowie j odpowiada i-tej literze `A` w słowie m, i-ta litera `B` w słowie j odpowiada i-tej literze `B` w słowie m itd. To pozwala bardzo łatwo skonstruować permutację przeprowadzającą j na m: otóż jeśli literą odpowiadającą literze j_i w słowie m jest m_k, to w permutacji p na pozycji i wstawiamy liczbę k, co oznacza tyle co: „chciałbym przestawić literę z pozycji i na pozycję k”. Przykładowo, dla słów j = `ABAAB`, m = `BABAA` permutacja przeprowadzająca pierwsze z nich na drugie ma postać $p = 2, 1, 4, 5, 3$.

[2] *Permutacją* nazywamy operację zmieniającą kolejność wyrazów zadanego ciągu. Dowolną permutację możemy przedstawić w postaci ciągu $(p_i)_{i=1}^{n}$ liczb całkowitych z zakresu od 1 do n, w którym każda z tych liczb występuje dokładnie raz. Zapis ten oznacza, że permutacja p dla każdego i przemieszcza wyraz z pozycji i na pozycję p_i.

Nietrudno jest napisać algorytm generujący permutację przeprowadzającą j na m przy użyciu tablicy stosów, wspomnianej w poprzednim rozdziale. Kiedy już będziemy znali tę permutację, możemy obliczyć liczbę inwersji w tej permutacji.

Definicja 2. Niech p będzie permutacją n liczb. *Inwersją* w permutacji p nazywamy dowolną takę parę indeksów $1 \leqslant a < b \leqslant n$, że $p_a > p_b$.

Innymi słowy, inwersją jest każda para elementów permutacji, która jest ustawiona w niewłaściwej kolejności w porównaniu do permutacji identycznościowej (posortowanej rosnąco). Na przykład permutacja identycznościowa $p = 1, 2, \ldots, n$ nie ma żadnych inwersji, a permutacja odwracająca kolejność $p = n, (n-1), \ldots, 1$ ma ich $\frac{n(n-1)}{2}$.

Przyszedł już czas na kluczowe spostrzeżenie: otóż szukany przez nas wynik jest równy właśnie liczbie inwersji w permutacji przeprowadzającej j na m! Dlaczego tak jest? Każde przesunięcie w słowie j powoduje zarazem odpowiadające mu przesunięcie (czyli podobnie jak w przypadku słów — zamianę dwóch sąsiednich liczb) w permutacji p, i odwrotnie. Słowo j zostaje ostatecznie przekształcone w m w momencie, w którym permutacja przeprowadzająca j na m jest posortowana. Ponieważ każde przesunięcie zmienia liczbę inwersji dokładnie o jeden (bo zmienia względne położenie dokładnie jednej pary sąsiednich elementów), więc przy przekształcaniu j na m trzeba wykonać co najmniej tyle przesunięć, ile wynosi liczba inwersji w p. Z drugiej strony, każdą permutację można posortować w następujący sposób: dopóki istnieje jakaś para sąsiednich elementów tworząca inwersję, wykonaj na nich przesunięcie. W momencie, w którym nie będzie już żadnej takiej pary, permutacja będzie posortowana[3]. Tak więc liczba inwersji jest nie tylko dolnym oszacowaniem na minimalną liczbę przesunięć przekształcających j w m, ale również i górnym, czyli jest równa szukanemu wynikowi.

Jak zliczać inwersje w permutacji?

Znanych jest kilka algorytmów pozwalających zliczać inwersje w permutacji. Najpopularniejszym rozwiązaniem w programach zawodników była pewna modyfikacja algorytmu Mergesort.

Algorytm Mergesort bazuje na operacji *scalenia* (ang. *merge*) dwóch posortowanych ciągów, która łączy je w ciąg posortowany w czasie liniowym. Z jej użyciem można skonstruować następujący algorytm sortowania:

```
1: function mergesort(ciąg)
2: begin
3:   if |ciąg| = 1 then return ciąg;
4:   (ciąg1, ciąg2) := podziel_na_połowy(ciąg);
5:   return merge(mergesort(ciąg1), mergesort(ciąg2));
6: end
```

Funkcja *podziel_na_połowy* w powyższym pseudokodzie „łamie ciąg na pół”, zwracając otrzymane w ten sposób dwie części, mniej więcej tego samego rozmiaru. Przy założeniu, że operacja $merge(ciąg1, ciąg2)$ działa w czasie $O(|ciąg1| + |ciąg2|)$,

[3] Mowa tu o algorytmie sortowania bąbelkowego.

cały algorytm Mergesort ma złożoność czasową $O(n \log n)$, gdzie n to długość sortowanego ciągu.

Okazuje się, że można tak zmodyfikować operację scalania, żeby oprócz łączenia dwóch posortowanych ciągów w nowy posortowany ciąg, obliczała liczbę takich inwersji (a, b), że a należy do pierwszego scalanego ciągu, zaś b do drugiego. Dla każdej pary wyrazów (a, b) wyjściowej permutacji p, będzie dokładnie jedno wywołanie funkcji *merge*(*ciąg1*, *ciąg2*) dla argumentów takich, że a należy do ciągu *ciąg1*, zaś b do ciągu *ciąg2* — nastąpi to w tym wywołaniu funkcji *mergesort*, w którym a i b zostaną rozdzielone do dwóch różnych połówek sortowanego ciągu. Jeżeli więc uda nam się wzbogacić funkcję *merge* w żądany sposób, to sumarycznie wyznaczy ona liczbę wszystkich inwersji w oryginalnym ciągu. Oto pseudokod takiej wzbogaconej operacji scalania:

```
1:  function merge(ciąg1, ciąg2)
2:  begin
3:    k1 := |ciąg1|;   k2 := |ciąg2|;
4:    i1 := 1;   i2 := 1;
5:    while i1 ⩽ k1 or i2 ⩽ k2 do begin
6:      if i1 > k1 then begin
7:        ciąg[i1 + i2 − 1] := ciąg2[i2];
8:        i2 := i2 + 1;
9:      end else if i2 > k2 then begin
10:       ciąg[i1 + i2 − 1] := ciąg1[i1];
11:       i1 := i1 + 1;
12:     end else if ciąg1[i1] < ciąg2[i2] then begin
13:       inwersje := inwersje + i2 − 1;
14:       ciąg[i1 + i2 − 1] := ciąg1[i1];
15:       i1 := i1 + 1;
16:     end else begin
17:       ciąg[i1 + i2 − 1] := ciąg2[i2];
18:       i2 := i2 + 1;
19:     end
20:   end
21:   return ciąg;
22: end
```

Dowód poprawności powyższej funkcji pozostawiamy Czytelnikowi jako ćwiczenie (warto przy tym pamiętać, że inwersje zliczamy tutaj tylko dla permutacji, a permutacje składają się z parami różnych elementów).

Istnieje też inny, podobny algorytm zliczający inwersje w permutacji, który zasugerował nam Marcin Pilipczuk. To podejście różni się od algorytmu Mergesort metodą dzielenia i scalania ciągu; poza tym, tym razem nie sortujemy elementów permutacji, a jedynie zliczamy inwersje metodą „dziel i zwyciężaj". Załóżmy, że dane wywołanie funkcji analizuje elementy permutacji należące do przedziału $[x, z]$ (początkowo $[1, n]$). Funkcja dzieląca ciąg dzieli go na pół nie według pozycji, ale według wartości, tzn. pierwszy ciąg będący rezultatem dzielenia składa się z liczb o wartościach z przedziału $[x, y]$, zaś drugi — z przedziału $[y + 1, z]$, przy czym y jest środkiem przedziału $[x, z]$.

Liczby inwersji w ramach tych podciągów obliczamy rekurencyjnie, natomiast liczbę inwersji pomiędzy ciągami wyznaczamy łatwo w czasie liniowym, korzystając z tego, że każdy element drugiego ciągu stanowi inwersję z każdym elementem pierwszego ciągu występującym dalej w oryginalnym ciągu. Dopracowanie szczegółów w tym podejściu pozostawiamy jako ćwiczenie. Złożoność czasowa tego rozwiązania to $O(n \log n)$, podobnie jak w przypadku algorytmu Mergesort.

Jeszcze inne, równie efektywne rozwiązanie problemu zliczania inwersji w permutacji jest przedstawione w artykule *Zliczamy inwersje* w czasopiśmie *Delta* [37].

Ciekawym spostrzeżeniem jest to, że skoro wynik w zadaniu jest równy liczbie inwersji w permutacji przeprowadzającej j na m, to podany we wcześniejszej sekcji algorytm wzorcowy daje zarazem rozwiązanie problemu zliczania inwersji w permutacji — również w czasie liniowo-logarytmicznym, ale za pomocą jeszcze innych narzędzi, tj. drzew przedziałowych.

Testy

Rozwiązania zawodników były sprawdzane z użyciem 10 zestawów testowych. Każdy zestaw składał się z jednego lub dwóch pojedynczych testów — patrz tabela poniżej.

Nazwa	n	Opis
lit1.in	4	mały test stworzony ręcznie
lit2.in	7	mały test stworzony ręcznie
lit3a.in	1 000	test losowy
lit3b.in	1 000	test wymagający dużej liczby przesunięć
lit4a.in	10 000	test losowy
lit4b.in	10 000	test wymagający dużej liczby przesunięć
lit5a.in	50 000	test losowy
lit5b.in	50 000	test wymagający dużej liczby przesunięć
lit6a.in	100 000	test losowy
lit6b.in	100 000	test wymagający dużej liczby przesunięć
lit7a.in	200 000	test losowy
lit7b.in	200 000	test wymagający dużej liczby przesunięć
lit8a.in	500 000	test losowy
lit8b.in	500 000	test wymagający dużej liczby przesunięć
lit9a.in	1 000 000	test losowy
lit9b.in	1 000 000	test wymagający dużej liczby przesunięć
lit10a.in	1 000 000	test losowy
lit10b.in	1 000 000	test wymagający dużej liczby przesunięć

Wojciech Śmietanka — Treść zadania, Opracowanie
Zbigniew Wojna — Program

Dostępna pamięć: 128 MB. *OI, Etap I, 17.10–14.11.2011*

Odległość

W tym zadaniu rozważamy **odległość** *między dodatnimi liczbami całkowitymi, zdefiniowaną w następujący sposób. Przez pojedynczą* **operację** *rozumiemy pomnożenie danej liczby przez liczbę pierwszą*[1] *lub podzielenie jej przez liczbę pierwszą (o ile dzieli się ona bez reszty). Odległość między liczbami a i b, oznaczana $d(a,b)$, to minimalna liczba operacji potrzebnych do przekształcenia liczby a w liczbę b. Na przykład, $d(69,42) = 3$.*

Zauważmy, że faktycznie funkcja d ma cechy odległości — dla dowolnych dodatnich liczb całkowitych a, b i c zachodzi:

- *odległość liczby od niej samej wynosi 0: $d(a,a) = 0$,*
- *odległość od a do b jest taka sama, jak od b do a: $d(a,b) = d(b,a)$, oraz*
- *spełniona jest nierówność trójkąta: $d(a,b) + d(b,c) \geqslant d(a,c)$.*

Dany jest ciąg n dodatnich liczb całkowitych $a_1, a_2, \ldots, a_n$. Dla każdej liczby a_i należy wskazać takie j, że $j \neq i$ oraz $d(a_i, a_j)$ jest minimalne.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($2 \leqslant n \leqslant 100\,000$). W kolejnych wierszach znajdują się liczby $a_1, a_2, \ldots, a_n$ ($1 \leqslant a_i \leqslant 1\,000\,000$), po jednej w wierszu.

W testach wartych łącznie 30% punktów zachodzi dodatkowy warunek $n \leqslant 1\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście dokładnie n wierszy, a w każdym z nich po jednej liczbie całkowitej. W i-tym wierszu należy wypisać **najmniejsze** *takie j, że: $1 \leqslant j \leqslant n$, $j \neq i$ oraz $d(a_i, a_j)$ jest minimalne.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
6	2
1	1
2	1
3	2
4	1
5	2
6	

[1] Przypomnijmy, że liczba pierwsza to taka liczba całkowita dodatnia, która ma dokładnie dwa różne dzielniki: jedynkę i siebie samą.

Rozwiązanie

Analiza problemu

Zacznijmy od przejścia na język teorii grafów. Rozważmy graf nieskierowany, w którym wierzchołkami są wszystkie liczby naturalne, a krawędzie odpowiadają pomnożeniu (równoważnie, podzieleniu) danej liczby przez liczbę pierwszą. Problem z zadania formułuje się teraz następująco: dla każdego z n zaznaczonych wierzchołków chcemy znaleźć w grafie najbliższy inny zaznaczony wierzchołek.

Wygodnie będzie, jeśli na wstępie usuniemy z podanego na wejściu ciągu liczb wszystkie powtórzenia. Faktycznie, dla każdej z powtarzających się liczb, jako wynik możemy od razu wskazać dowolne inne wystąpienie takiej samej liczby w ciągu. Odtąd będziemy zakładać, że startowy ciąg nie zawiera powtórzeń.

Jak duży jest nasz graf?

Oznaczmy przez M największą liczbę z wejścia. W grafie będziemy mieli M wierzchołków, z czego n zaznaczonych. Krawędzi w takim grafie jest potencjalnie rzędu M^2, czyli nawet 10^{12}, czyli bardzo dużo. W przypadku naszego grafu sytuacja jest jednak trochę lepsza, dzięki temu, że każda z krawędzi w grafie reprezentuje, w szczególności, podzielenie liczby przez jakąś liczbę pierwszą.

Wydaje się, że takich krawędzi nadal może być bardzo dużo. Najprostsze oszacowanie daje $M \cdot \pi(M)$ krawędzi, gdzie $\pi(M)$ jest liczbą liczb pierwszych mniejszych od M. Jest jednak dużo lepiej: krawędzi jest dokładnie tyle, ile w sumie dzielników pierwszych wszystkich liczb od 1 do M. Przy założeniu, że $M \leqslant 10^6$, mamy co najwyżej 7 różnych dzielników pierwszych, gdyż

$$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 = 9\,699\,690 > 10^6.$$

Czyli wszystkich krawędzi jest co najwyżej $7M$. Wiedząc, że:

$$\sum_{p \leqslant M,\, p \in P} \frac{1}{p} \approx \ln \ln M,$$

wnioskujemy, że liczba krawędzi grafu jest asymptotycznie rzędu $O(M \ln \ln M)$, czyli dostatecznie mało, aby można było przejrzeć cały graf.

Jak przejrzeć cały graf?

Standardowe algorytmy służące do obliczania odległości między wszystkimi parami wierzchołków w naszym przypadku zawiodą, gdyż potrzebują one czasu co najmniej kwadratowego ze względu na liczbę wierzchołków. Dla nas czas $\Omega(n^2)$ czy $\Omega(M^2)$ to zdecydowanie za dużo. Warto jednak zauważyć pewien ciekawy fakt:

Fakt 1. *Każdą ścieżkę między liczbami a i b jesteśmy w stanie opisać następująco:*

- *dzielimy a przez wszystkie nadmiarowe dzielniki pierwsze*
- *mnożymy a przez wszystkie brakujące dzielniki pierwsze.*

Powyższy fakt ma w rzeczywistości istotny związek z liczbą $\mathrm{NWD}(a, b)$. Do tej liczby schodzimy w dół, dzieląc, i od tej liczby zaczynamy wchodzić do góry, mnożąc.

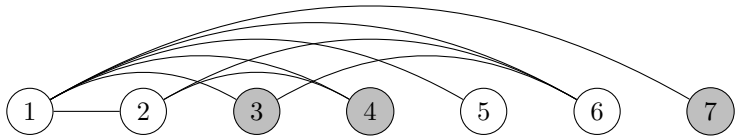
Możemy już teraz zaproponować bardziej efektywne rozwiązanie. Najpierw, startując ze wszystkich zaznaczonych wierzchołków, schodzimy w dół z informacją, że gdzieś blisko jest zaznaczona liczba. W ten sposób dla wszystkich liczb zapamiętamy najbliższe zaznaczone liczby będące ich wielokrotnościami. Następnie wykonujemy drugą fazę, w której przekazujemy informacje o najbliższych zaznaczonych liczbach do góry. Zauważmy, że dla każdego wierzchołka wystarczy przechowywać tu dane dotyczące dwóch najbliższych zaznaczonych wierzchołków. A czemu tak? Otóż pamiętanie informacji tylko o jednym wierzchołku to za mało, gdyż dla każdej zaznaczonej liczby najbliższą zaznaczoną jest ona sama. Dlatego warunek z treści zadania możemy sformułować jako: dla każdego zaznaczonego wierzchołka, wyznacz indeks drugiego najbliższego mu zaznaczonego wierzchołka.

Całe rozwiązanie wygląda następująco.

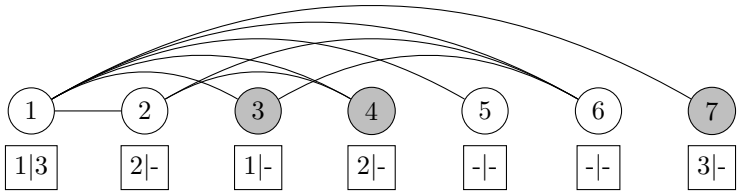
1. Generujemy listę wszystkich liczb pierwszych nie większych niż M.
2. Dla każdej liczby od 2 do M obliczamy jakiś dzielnik pierwszy.
3. Dla każdej liczby od 1 do M obliczamy dwie najbliższe zaznaczone liczby, używając tylko dzieleń, czyli zejść w dół od zaznaczonych liczb.
4. Idziemy w górę, czyli dla każdej liczby przemnażamy ją przez wszystkie dzielniki pierwsze, tak aby nie przekroczyć M, i propagujemy w górę informacje o najbliższych dzielnikach.

Działanie algorytmu na przykładzie

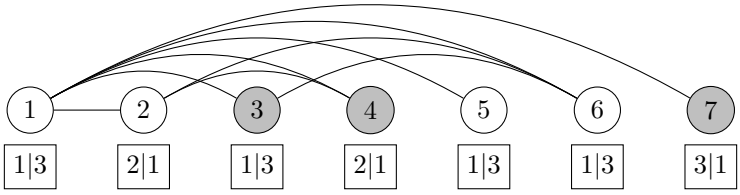
Poniższy rysunek przedstawia graf zbudowany dla $M = 7$. Naszym ciągiem na wejściu niech będzie $3, 4, 7$ (szare wierzchołki na rysunku).



W pierwszej fazie algorytmu przekazujemy w dół informacje o dwóch najbliższych zaznaczonych wierzchołkach. W ramkach podane są indeksy zaznaczonych elementów, tzn. indeks trójki to 1, indeks czwórki to 2, a indeks siódemki to 3.



A teraz druga faza — przejście w górę.



Złożoność

Złożoność czasowa podanego dwufazowego przeszukiwania grafu jest liniowa względem rozmiaru grafu, czyli rzędu $O(M \ln \ln M)$. Należy zauważyć, że nie trzeba trzymać w pamięci całego grafu G. Wystarczy dla każdej liczby mieć obliczony pewien jej dzielnik pierwszy (do schodzenia w dół) i osobno pamiętać listę wszystkich liczb pierwszych (do wchodzenia w górę). Zarówno listę wszystkich liczb pierwszych nieprzekraczających M, jak i przykładowe dzielniki pierwsze poszczególnych liczb od 2 do M można obliczyć w czasie $O(M \ln \ln M)$ za pomocą sita Eratostenesa[1]. Dodajmy dla pełności, że początkowe usuwanie powtórzeń z wyjściowego ciągu możemy wykonać w czasie i pamięci $O(M)$, wykorzystując M-elementową tablicę kubełków (list). Ostatecznie, całe rozwiązanie ma złożoność czasową $O(M \ln \ln M)$, a pamięciową $O(M)$.

Testy

Przygotowanych zostało 10 zestawów testowych. Jeśli dana grupa składa się z więcej niż jednego testu, to pierwszy test jest testem poprawnościowym, co może np. oznaczać, że odległości do najbliższych liczb są większe aniżeli w losowym teście. W tych testach parametr n jest zazwyczaj stosunkowo niewielki.

Nazwa	n	M	Opis
odl1.in	50	95	test losowy
odl2.in	345	698	potęgi liczb pierwszych

[1]Przykład takiego wykorzystania sita Eratostenesa można znaleźć w opracowaniu zadania *Zapytania* z XIV Olimpiady Informatycznej [14].

Nazwa	n	M	Opis
odl3a.in	584	3 499	trochę maksymalnych potęg liczb pierwszych, trochę losowy
odl3b.in	1 000	3 500	losowy test wydajnościowy
odl4a.in	11 884	80 649	liczby o nieparzystej i większej niż 4 sumie wykładników w rozkładzie
odl4b.in	21 234	47 288	losowy test wydajnościowy
odl5a.in	13 459	150 000	kilka liczb, od których jest daleko do najbliższego elementu, reszta losowa
odl5b.in	50 001	150 001	losowy test wydajnościowy
odl6a.in	34 209	399 989	mniejsze liczby: losowe, większe liczby: maksymalne potęgi liczb pierwszych
odl6b.in	73 900	234 564	losowy test wydajnościowy
odl7a.in	36 118	756 432	dużo liczb, od których odległość do najbliższej to 2, trochę mniej tych, od których odległość to 3
odl7b.in	86 023	803 819	losowy test wydajnościowy
odl8a.in	63 759	999 999	dużo liczb o odległości co najmniej 2, nieliczne o dużo większych odległościach do im najbliższych
odl8b.in	98 837	853 983	losowy test wydajnościowy
odl9a.in	1 482	999 810	losowe odległości między 1 a 5
odl9b.in	99 048	1 000 000	losowy test wydajnościowy
odl10a.in	1 176	1 000 000	potęgi liczb naturalnych o wykładnikach większych niż 1
odl10b.in	100 000	999 996	losowy test wydajnościowy
odl10c.in	100 000	999 999	maksymalne wejście

Alan Kutniewski
Treść zadania, Opracowanie

Zbigniew Wojna
Program

Dostępna pamięć: 128 MB.

OI, Etap I, 17.10–14.11.2011

Randka

Bajtazar jest strażnikiem przyrody i pracuje w Jaskini Strzałkowej — znanym miejscu schadzek zakochanych par. Jaskinia ta składa się z n komór połączonych jednokierunkowymi korytarzami. W każdej komorze dokładnie jeden wychodzący z niej korytarz jest oznaczony strzałką. Każdy korytarz prowadzi bezpośrednio z jednej komory do pewnej (niekoniecznie innej) komory.

Notorycznie zdarza się, że zakochane pary, które umawiają się na randki w Jaskini Strzałkowej, zapominają dokładnie ustalić miejsce spotkania i nie mogą się odnaleźć. W przeszłości prowadziło to do wielu nieporozumień i pomyłek... Od czasu, gdy w każdej komorze zainstalowano telefon alarmowy łączący z dyżurnym strażnikiem przyrody, głównym zajęciem strażników stało się pomaganie zakochanym parom w odnajdowaniu się.

Strażnicy wypracowali następującą metodę. Wiedząc, w których komorach znajdują się zakochani, mówią każdemu z nich, ile razy, odpowiednio, powinien przejść z komory do komory korytarzem oznaczonym strzałką, aby oboje mogli się spotkać. Przy tym, zakochani bardzo chcą spotkać się jak najszybciej — zależy im przecież, aby razem miło spędzać czas, a nie samotnie przemierzać korytarze. Strażnicy starają się podawać zakochanym parom takie liczby, aby ich maksima były możliwie jak najmniejsze.

Bajtazar jest już zmęczony ciągłym pomaganiem zakochanym i poprosił Cię o napisanie programu, który by to usprawnił. Program ten, na podstawie opisu Jaskini Strzałkowej oraz aktualnego położenia k zakochanych par, powinien wyznaczyć k par liczb x_i i y_i, takich że:

- *jeżeli i-ta para zakochanych przejdzie odpowiednio: on x_i, a ona y_i korytarzami oznaczonymi strzałkami, to spotkają się w jednej komorze jaskini,*
- *$max(x_i, y_i)$ jest jak najmniejsze,*
- *w drugiej kolejności $min(x_i, y_i)$ jest jak najmniejsze,*
- *jeżeli rozwiązanie wciąż nie jest jednoznaczne, to kobieta powinna pokonywać mniejszy dystans.*

Może się tak zdarzyć, że takie liczby x_i i y_i nie istnieją — wówczas przyjmujemy, że $x_i = y_i = -1$.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie dodatnie liczby całkowite n i k ($1 \leqslant n \leqslant 500\,000$, $1 \leqslant k \leqslant 500\,000$), oddzielone pojedynczym odstępem i określające liczbę komór w Jaskini Strzałkowej i liczbę zakochanych par, które chcą się odnaleźć. Komory są ponumerowane od 1 do n, natomiast zakochane pary są ponumerowane od 1 do k.

W drugim wierszu wejścia znajduje się n liczb całkowitych dodatnich, pooddzielanych pojedynczymi odstępami: i-ta liczba w tym wierszu określa numer komory, do której prowadzi korytarz oznaczony strzałką wychodzący z komory numer i.

W kolejnych k wierszach znajdują się kolejne zapytania zakochanych par. Każde zapytanie składa się z dwóch liczb całkowitych dodatnich oddzielonych pojedynczym odstępem — oznaczają one numery komór, w których znajduje się dana para zakochanych — najpierw on, a potem ona.

W testach wartych łącznie 40% punktów zachodzą dodatkowe warunki $n \leqslant 2\,000$ *oraz* $k \leqslant 2\,000$*.*

Wyjście

Twój program powinien wypisać na standardowe wyjście dokładnie k wierszy, po jednym dla każdej pary zakochanych z wejścia: i-ty wiersz wyjścia powinien opisywać wynik dla i-tej pary z wejścia. Każdy wiersz wyjścia powinien składać się z dwóch liczb całkowitych x_i, y_i, *oddzielonych pojedynczym odstępem.*

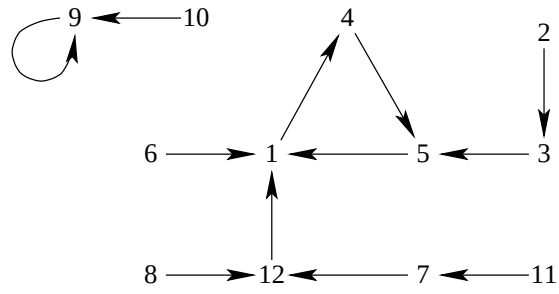
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2 3
1 2
2 2
0 1
-1 -1
```



Rozwiązanie

Analiza problemu

Na początku zastanówmy się, jaką postać ma graf opisany w tym zadaniu. Jest on skierowany, a z każdego wierzchołka wychodzi dokładnie jedna krawędź. Nasz graf składa się zatem z pewnej liczby cykli, do których mogą był dołączone drzewa[1]. W dalszym opisie przyjmiemy, że każdy wierzchołek cyklu stanowi korzeń pewnego drzewa — w niektórych przypadkach drzewo to składa się tylko z tego wierzchołka.

Następnie mamy daną listę par wierzchołków w tym grafie. Dla każdej pary wierzchołków musimy znaleźć taką *optymalną* parę liczb x, y, że po przejściu x kroków

[1]Ciekawe własności takich grafów były niejednokrotnie wykorzystywane w rozwiązaniach zadań olimpijskich, np. w zadaniu *Mafia* z XV Olimpiady Informatycznej [15] i w zadaniu *Szpiedzy* z XI Olimpiady Informatycznej [11].

z pierwszego wierzchołka i y kroków z drugiego dojdziemy do tego samego wierzchołka. Szukana optymalna para x, y powinna przede wszystkim minimalizować $\max(x, y)$, w drugiej kolejności — $\min(x, y)$, a w trzeciej kolejności — parametr y.

Pomyślmy teraz, gdzie może leżeć wierzchołek, w którym nastąpi spotkanie. Możliwe są trzy przypadki:

1. Wierzchołki leżą na jednym drzewie i na nim też się spotkają.
2. Wierzchołki leżą na różnych drzewach, których korzenie znajdują się na jednym cyklu. Wówczas spotkanie nastąpi na tym cyklu.
3. Wierzchołki nie mogą się spotkać.

Rozwiązanie siłowe

Na podstawie powyższych obserwacji możemy skonstruować pierwsze rozwiązanie. Zaznaczamy wszystkie wierzchołki, do których może dojść pierwsza osoba. Następnie drugą osobę przesuwamy po strzałkach, aż dojdzie do jednego z wierzchołków, które zaznaczyliśmy. Później robimy to samo, odwracając role. Jako wynik podajemy lepsze z dwóch znalezionych miejsc spotkania (używając zadanego kryterium porównywania wyników). Należy pamiętać o przypadku, w którym osoby nie mogą się spotkać.

Dlaczego otrzymany w ten sposób wynik jest optymalny? Wróćmy do określonych wcześniej przypadków.

W pierwszym przypadku, wykonując powyższy algorytm, znajdziemy najbliższego wspólnego przodka danej pary wierzchołków w drzewie. Jedynymi innymi potencjalnymi miejscami spotkań są wierzchołki bliższe korzenia lub wierzchołki leżące na cyklu. Jednak żeby do nich dojść, trzeba zwiększyć zarówno x, jak i y, co z oczywistych względów nie da nam lepszego wyniku.

Drugi przypadek jest trochę bardziej skomplikowany. Potencjalne miejsca spotkań leżą na cyklu. Jeśli więc któryś wierzchołek z pary nie leży na cyklu, to musi przejść w górę drzewa, aż dojdzie do cyklu. Teraz są dwie możliwości: albo pierwszy wierzchołek przejdzie po cyklu do drugiego, albo na odwrót. Tym dwóm przypadkom odpowiadają dwa przebiegi naszego algorytmu. Zatem bierzemy pod uwagę dwa potencjalne miejsca spotkań: pierwszy wierzchołek leżący na cyklu należący do ścieżki wychodzącej z pierwszego wierzchołka pary oraz analogiczny wierzchołek należący do ścieżki wychodzącej z drugiego wierzchołka z pary. Każdy inny wierzchołek cyklu może być tylko gorszym kandydatem na miejsce spotkania, ponieważ ścieżki z obydwu wierzchołków z pary do dowolnego niewybranego przez nas wierzchołka na cyklu prowadzą, odpowiednio, przez wybrane przez nas miejsca.

Złożoność czasowa tego rozwiązania to $O(n \cdot k)$.

Wnioski z rozwiązania siłowego

Analizując rozwiązanie siłowe, poczyniliśmy bardzo ważne spostrzeżenia.

1. Jeśli wierzchołki z pary leżą na jednym drzewie, to wynikiem jest ich najniższy wspólny przodek (tzw. *LCA*, ang. *lowest common ancestor*).

2. W przeciwnym razie jedynymi kandydatami na optymalne miejsce spotkania są pierwsze wierzchołki na ścieżkach wychodzących z danej pary wierzchołków, które leżą na cyklu.

Optymalizacje

Optymalne miejsca spotkań zależą od przypadku, z którym mamy do czynienia. Musimy więc umieć szybko go określać. Dla każdego wierzchołka znajdujemy cykl, do którego można z niego dojść, oraz drzewo, w którym się znajduje. Zrobimy to w dwóch krokach.

W pierwszym kroku chcemy znaleźć cykl. W tym celu wybieramy dowolny wierzchołek i idziemy po strzałkach. Musimy w końcu dojść do odwiedzonego wcześniej wierzchołka. To będzie oznaczało, że znaleźliśmy cykl; teraz wystarczy przejść po nim, zaznaczając go. Można to zrobić przy pomocy algorytmu przedstawionego poniżej.

```
1:  for i := 1 to n do begin
2:      odwiedzony[i] := false;
3:      cykl[i] := −1;
4:  end
5:  wierzchołek := 1;
6:  while not odwiedzony[wierzchołek] do begin
7:      odwiedzony[wierzchołek] := true;
8:      wierzchołek := następny[wierzchołek];
9:  end
10: { W tym miejscu wierzchołek leży na cyklu }
11: ostatni_wierzchołek := wierzchołek;
12: do
13:     cykl[wierzchołek] := nr_cyklu;
14:     wierzchołek := następny[wierzchołek];
15: while wierzchołek ≠ ostatni_wierzchołek;
```

W drugim kroku zaznaczamy drzewa, które wchodzą do cyklu znalezionego w pierwszym kroku. Przechodzimy po wszystkich wierzchołkach należących do cyklu. Z każdego z nich idziemy w przeciwną stronę do tej, którą pokazują strzałki, do wierzchołków, które nie mają jeszcze przypisanego cyklu. Innymi słowy, przeszukujemy wszerz (algorytm BFS) graf odwrócony. Każdemu wierzchołkowi zapisujemy numer wierzchołka cyklu, z którego przyszliśmy do tego wierzchołka, jako numer jego drzewa. Wierzchołki leżące na cyklu są korzeniami tych drzew.

Po wykonaniu tych dwóch kroków mamy oznaczony jeden cykl oraz wszystkie jego drzewa. Musimy powtarzać ten algorytm, dopóki istnieją nieoznaczone wierzchołki, ignorując wierzchołki, które już zostały oznaczone.

Wszystkie wstępne obliczenia działają w złożoności $O(n)$. Obliczona za ich pomocą struktura danych pozwala łatwo określać, dla każdego zapytania, z którym przypadkiem mamy do czynienia. Odtąd skupimy się na rozwiązywaniu obu przypadków osobno.

Pierwszy przypadek: LCA

Zakładamy, że obydwa wierzchołki z pary leżą na jednym drzewie i szukamy ich najniższego wspólnego przodka. To już jest standardowy problem. Algorytm znajdujący LCA w czasie $O(\log n)$ (po wstępnych obliczeniach o złożoności czasowej i pamięciowej $O(n \log n)$) można znaleźć, na przykład, w opisie rozwiązania zadania *Komiwojażer Bajtazar* z IX Olimpiady Informatycznej [9].

Drugi przypadek: wynik leży na cyklu

W tym przypadku mamy dwóch kandydatów na optymalne miejsce spotkania. Musi nim być korzeń drzewa, na którym leży jeden z danych wierzchołków.

Przede wszystkim, obydwa wierzchołki muszą znać swoją odległość od korzenia. Takie wartości możemy łatwo zapamiętywać podczas wstępnych obliczeń (algorytm BFS).

Teraz musimy jeszcze obliczyć odległość na cyklu z pierwszego korzenia do drugiego, a także z drugiego do pierwszego. Żeby to zrobić, zapamiętujemy długość cyklu oraz numerujemy kolejno jego wierzchołki. Odległość między dwoma korzeniami na cyklu to różnica ich numerów, gdy jest dodatnia, lub długość cyklu pomniejszona o tę różnicę w przeciwnym przypadku.

Podsumowanie

W celu określenia przypadku wykonujemy wstępne obliczenia działające w czasie $O(n)$. Potrzebujemy także czasu $O(n \log n)$ na wstępne obliczenia związane z wyznaczaniem LCA par wierzchołków. Następnie k razy znajdujemy optymalne miejsce spotkania. Obliczanie LCA działa w czasie $O(\log n)$, a obliczanie wyniku na cyklu — w czasie $O(1)$.

Cały algorytm działa więc w czasie $O((n + k) \log n)$ i pamięci $O(n \log n)$.

Testy

Rozwiązania zawodników sprawdzano za pomocą 10 zestawów testowych. Testy 1 i 2 to proste testy generowane ręcznie. Każdy z zestawów 3–10 składał się z trzech testów następujących typów:

a — jedna bardzo długa gałąź z losowymi zaburzeniami. Na takich testach wolno działają te rozwiązania, w których wyznaczanie LCA jest liniowe.

b — jeden bardzo duży cykl z losowymi zaburzeniami. Na takich testach wolno działają te rozwiązania, w których wyznaczanie najkrótszej ścieżki na cyklu jest liniowe.

c — test zawierający różne trudne przypadki. Mały graf i kilka zapytań są wszędzie takie same, reszta wierzchołków i zapytań jest wybrana losowo.

Na następnej stronie można znaleźć tabelę zawierającą parametry poszczególnych testów.

Nazwa	**n**	**k**
ran1.in	3	5
ran2.in	9	4
ran3abc.in	1 000	1 000
ran4abc.in	2 000	2 000
ran5abc.in	50 000	50 000
ran6abc.in	100 000	100 000
ran7abc.in	200 000	200 000
ran8abc.in	500 000	250 000
ran9abc.in	500 000	500 000
ran10abc.in	500 000	500 000

Michał Włodarczyk
Treść zadania, Opracowanie

Igor Adamski
Program

Dostępna pamięć: 64 MB.

OI, Etap I, 17.10–14.11.2011

Studnia

Bajtazar wybrał się na wyprawę wzdłuż Suchej Rzeki, która przecina Pustynię Bajtocką. Niestety Sucha Rzeka wyschła, a Bajtazarowi skończyła się woda. Jedynym ratunkiem dla Bajtazara jest wykopanie studni na dnie wyschniętego koryta rzeki i dokopanie się do wody.

Bajtazar postanowił dobrze przemyśleć, co ma zrobić, zanim weźmie się za kopanie — wie, że jeśli opadnie z sił, a nie dokopie się do wody, to będzie miał skrajnie małe szanse na przetrwanie. Udało mu się określić, na jakiej głębokości pod dnem rzeki zalega woda. Wie też, na ile kopania starczy mu sił. Boi się tylko, żeby w czasie kopania nie osunęła się ziemia, gdyż może go pogrzebać żywcem. Bajtazar przesłał Ci (przez telefon satelitarny) opis topografii koryta rzeki. Poprosił Cię o wyznaczenie planu, gdzie ma kopać, tak aby dokopać się do wody, zanim opadnie z sił, a równocześnie, żeby zbocza w wykopie były jak najłagodniejsze. Bajtazar czeka na Twoją pomoc!

Wejście

W pierwszym wierszu standardowego wejścia są zapisane dwie dodatnie liczby całkowite n oraz m ($1 \leqslant n \leqslant 1\,000\,000$, $1 \leqslant m \leqslant 10^{18}$), oddzielone pojedynczym odstępem. W drugim wierszu znajduje się n dodatnich liczb całkowitych $x_1, x_2, \ldots, x_n$ ($1 \leqslant x_i \leqslant 10^9$), pooddzielanych pojedynczymi odstępami.

Bajtazarowi zostało sił na m ruchów łopatą. Liczby $x_1, x_2, \ldots, x_n$ stanowią opis topografii koryta Suchej Rzeki, zdatnego do kopania studni. Liczby te reprezentują grubość warstwy piasku ponad poziomem wody gruntowej, w kolejnych miejscach, co metr wzdłuż koryta rzeki. Jednym ruchem łopaty Bajtazar może wybrać tyle piachu, aby jedną z liczb x_i zmniejszyć o 1. Jeżeli którakolwiek z liczb x_i, powiedzmy x_k, zmniejszy się do 0, będzie to oznaczać, że Bajtazar dokopał się do wody. Poza dokopaniem się do wody w co najmniej jednym punkcie koryta rzeki, Bajtazarowi zależy na tym, aby na końcu następująca liczba z, charakteryzująca nachylenie piaszczystych zboczy:

$$z = \max_{i=1,2,\ldots,n-1} |x_i - x_{i+1}|,$$

była jak najmniejsza. Jeżeli istnieje wiele poprawnych wartości liczby k, reprezentującej miejsce, w którym Bajtazar powinien dokopać się do poziomu wody, Twój program powinien wypisać dowolną z nich. Możesz przyjąć, że poza miejscami $1, 2, \ldots, n$ na wszystkich głębokościach znajduje się lita skała oraz że Bajtazar zawsze będzie miał wystarczająco dużo siły, żeby w którymś miejscu dokopać się do wody.

W testach wartych co najmniej 35% punktów zachodzi dodatkowy warunek $n \leqslant 10\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście dwie liczby całkowite oddzielone pojedynczym odstępem: miejsce k, w którym Bajtazar powinien dokopać się do wody, oraz najmniejszą możliwą wartość liczby z.

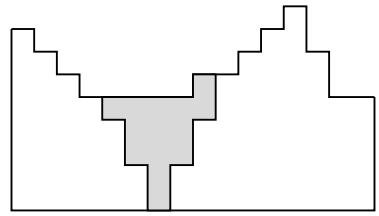
Przykład

Dla danych wejściowych:

```
16 15
8 7 6 5 5 5 5 6 6 7 8 9 7 5 5
```

poprawnym wynikiem jest:

```
1 2
```



Na powyższym rysunku prawidłowy wykop Bajtazara oznaczono szarym kolorem.

Rozwiązanie

Pierwsze podejście

Rozważania rozpoczniemy od skonstruowania najprostszego rozwiązania, które będziemy stopniowo ulepszać, aż dojdziemy do efektywnego algorytmu.

Na początku ustalmy k i z. Chcemy sprawdzić, czy można dokopać się do wody w punkcie k, tak aby nachylenie zbocza nie przekroczyło z. Dla tych parametrów, przez (y_i) oznaczmy ciąg opisujący topografię koryta rzeki po dokopaniu się do wody przy minimalnej liczbie ruchów łopatą (niedługo okaże się, że jest on wyznaczony jednoznacznie). Chcemy stwierdzić, czy:

$$\sum_{i=1}^{n}(x_i - y_i) \leqslant m.$$

Po pierwsze, wiemy, że $y_k = 0$, bo w tym miejscu dokopaliśmy się do wody. Dla $i \neq k$ pozostawmy $y_i = x_i$ i postarajmy się teraz wygładzić teren (tzn. usunąć część piasku tak, aby nachylenie nie przekraczało z). Wyobraźmy sobie, że idziemy w prawo i kiedy natrafiamy na próg o wysokości większej od z, pomniejszamy go. Następnie powtarzamy tę czynność, idąc w lewo. Poniżej przedstawiamy pseudokod tej procedury i dowód poprawności (ciągowi (y_i) odpowiada w kodzie tablica $y[]$):

```
1: for i := 1 to n − 1 do
2:     if y[i + 1] > y[i] + z then
3:         y[i + 1] := y[i] + z;
4: for i := n downto 2 do
5:     if y[i − 1] > y[i] + z then
6:         y[i − 1] := y[i] + z;
```

Lemat 1. Dla ciągu (y_i) po wykonaniu powyższej procedury zachodzi:

$$\forall_{i=1,\ldots,n-1} \quad |y_i - y_{i+1}| \leqslant z \tag{1}$$

i każdy inny ciąg (v_i) o tej własności, $0 \leqslant v_i \leqslant x_i$, $v_k = 0$, spełnia:

$$\forall_{i=1,\ldots,n} \quad v_i \leqslant y_i.$$

Dowód: Załóżmy, że po wykonaniu podanego algorytmu nierówność (1) nie jest spełniona, tzn. dla pewnego j zachodzi $y_j > y_{j+1}+z$ lub $y_{j+1} > y_j+z$. Pierwsza możliwość jest wykluczona, ponieważ w drugiej części procedury dla $i = j + 1$ poprawiliśmy y_j, tak aby zachodziło $y_j \leqslant y_{j+1} + z$, a później żadna z tych liczb nie była zmieniana. W drugim przypadku natomiast wiemy, że po przejściu w prawo było $y_{j+1} \leqslant y_j + z$. Idąc w lewo, nie mogliśmy zwiększyć y_{j+1}. Jeśli zaś y_j zostało zmienione, to spełnia równość $y_j = y_{j+1} + z$, co jest sprzeczne z założeniem.

Drugą część tezy dowiedziemy indukcyjnie. Dokładniej, pokażemy, że dla dowolnego ciągu $(x'_i)_{i=1}^n$ (odpowiadającego ciągowi $(x_i)_{i=1}^n$ z ustawionym $x_k = 0$) i ciągów $(y_i)_{i=1}^n$ i $(v_i)_{i=1}^n$, ograniczonych z góry przez ciąg x'_i i spełniających nierówność typu (1), przy czym ciąg y_i jest skonstruowany za pomocą podanego wyżej algorytmu, dla każdego i zachodzi $v_i \leqslant y_i$. Baza indukcji jest trywialna. Przypuśćmy, że teza zachodzi dla wszystkich ciągów długości $n - 1$. Pokażemy, że stąd wynika teza dla ciągów długości n.

Zauważmy, że $v_2, y_2 \leqslant \min(x'_2, x'_1 + z)$. Co więcej, gdybyśmy w podanym wyżej algorytmie wystartowali od ciągu $\min(x'_2, x'_1 + z), x'_3, \ldots, x'_n$, skonstruowalibyśmy właśnie ciąg $y_2, y_3, \ldots, y_n$. Stosując w tym miejscu założenie indukcyjne, dostajemy $v_i \leqslant y_i$ dla $i > 1$. Wreszcie

$$v_1 \leqslant \min(x'_1, v_2 + z) \leqslant \min(x'_1, y_2 + z) = y_1,$$

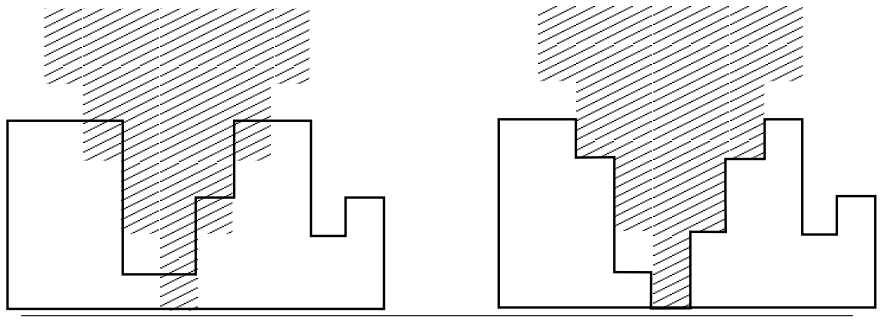
co dowodzi tezy indukcyjnej. ■

Z powyższego lematu wynika, że nasza procedura minimalizuje liczbę ruchów łopatą potrzebnych do wygładzenia terenu. Wiemy zatem, jak sprawdzać w czasie $O(n)$, czy możemy dokopać się do wody dla zadanych k oraz z. Oczywiście, jeśli możemy dokopać się do wody w punkcie k z nachyleniem nie większym niż z, to dla $z + 1$ również dostaniemy pozytywną odpowiedź. Ponadto dla $z = \max x_i$ nie trzeba nic wygładzać — wystarczy dokopać się do wody w najniższym punkcie, a w treści zadania zagwarantowano, że jest to możliwe. To pozwala znajdować najmniejsze z przy pomocy wyszukiwania binarnego, co stanowi klucz do istotnego przyspieszenia obliczeń. Zaprezentowane rozwiązanie działa w czasie $O(n^2 \log(\max x_i))$ i pozwalało na zawodach zdobyć trochę punktów, ale niestety nie radziło sobie z największymi testami.

Rozwiązanie wzorcowe

Aby pozbyć się złożoności kwadratowej, musimy sprytniej obliczać liczbę potrzebnych ruchów łopatą. Cofnijmy się do miejsca, w którym przyjęliśmy $y_k = 0$, i załóżmy, że z jest ustalone. Wiemy, że $y_{k-1}, y_{k+1} \leqslant z$, bo inaczej przekroczylibyśmy dozwolone nachylenie. Indukcyjnie pokazujemy, że $y_{k-j}, y_{k+j} \leqslant jz$ dla $j = 1, 2, \ldots$ Z tego wynika, że ze studni musimy usunąć cały piasek zawarty w „odwróconej piramidzie" o środku w k i nachyleniu z (patrz rys. 1).

Czy to wystarczy do zapewnienia bezpieczeństwa? Otóż nie; na rysunku widać, że problem może pojawić się z dala od pozycji k lub zbocze w okolicy pozycji k może mieć nieregularny kształt.



Rys. 1: Zakreskowany teren to odwrócona piramida dla $z = 2$. Po lewej stronie widać pierwotną topografię, po prawej zaś teren po usunięciu piasku z piramidy.

Jednak po chwili zastanowienia można zaryzykować tezę, że trudności te nie miałyby miejsca, gdyby nachylenie studni przed rozpoczęciem kopania nie przekraczało z. Nasz wysiłek umysłowy z pierwszego rozdziału nie pójdzie na marne — dzięki lematowi 1 wiemy, jak optymalnie wygładzić ciąg (x_i) dla zadanego z, a potem spróbujemy sztuczki z piramidą. Najpierw jednak musimy przekonać się, czy to aby na pewno wystarczy.

Lemat 2. Wygładzenie ciągu (x_i) do nachylenia z oraz usunięcie piasku zawartego w piramidzie o środku w k i nachyleniu z zadaje bezpieczny wykop do punktu k przy minimalnej liczbie ruchów łopatą.

Dowód: Przez (y_i) oznaczmy ciąg po wygładzeniu, zaś przez (u_i) — ciąg końcowy. Usunięcie piasku z piramidy oznacza, że na pozycji i otrzymamy wyraz $u_i = \min(y_i, |i-k| \cdot z)$. Upewnimy się teraz, że dla każdego i zachodzi $u_i \leqslant u_{i+1} + z$. Bez trudu dostajemy:

$$\begin{aligned} u_i &\leqslant y_i \leqslant y_{i+1} + z, \\ u_i &\leqslant |i-k| \cdot z \leqslant |(i+1)-k| \cdot z + z, \end{aligned}$$

a z połączenia tych dwóch nierówności mamy

$$u_i \leqslant \min(y_{i+1} + z,\, |(i+1)-k| \cdot z + z) = \min(y_{i+1},\, |(i+1)-k| \cdot z) + z = u_{i+1} + z.$$

Symetryczną nierówność dowodzimy w taki sam sposób.

Niech teraz (v_i) będzie dowolnym ciągiem zadającym bezpieczny wykop przy założeniach lematu. Jest on, w szczególności, ciągiem wygładzającym (x_i), więc na mocy lematu 1 wiemy, że $v_i \leqslant y_i$ dla każdego i. W połączeniu z nierównością $v_i \leqslant |i-k| \cdot z$, otrzymujemy:

$$v_i \leqslant \min(y_i, |i-k| \cdot z) = u_i.$$

To dowodzi, że liczby ruchów łopatą nie da się poprawić. ■

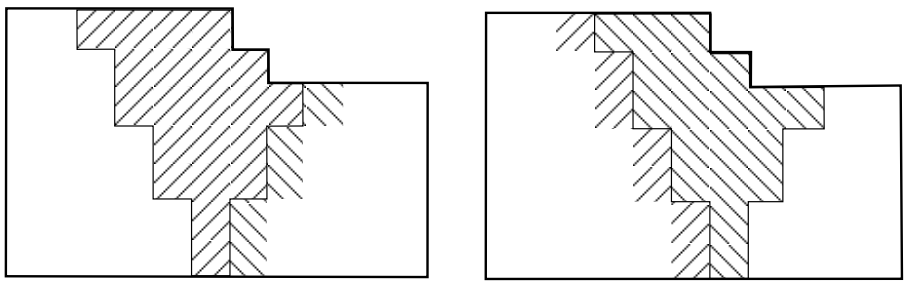
Daje to następujący pomysł na lepszy algorytm: wyszukujemy binarnie z i dla ustalonego z wygładzamy teren, po czym obliczamy, ile piasku mieści się w odwróconych piramidach dla różnych k. Chcemy rozstrzygnąć, czy dla pewnego k łączna

liczba ruchów ruchów łopatą nie przekracza m. Pozostaje tylko wymyślić sposób na szybkie obliczanie zawartości piramid.

Studnia pełna piramid

Na początek zauważmy, że dla $z = 0$ wystarczy sprawdzić, czy $\sum_{i=1}^{n} x_i \leqslant m$. Nieuwzględnienie tego przypadku może prowadzić do błędów w dalszej części rozważań.

Przyjmijmy teraz, że $z > 0$. Niech ciąg y_i oznacza wygładzony ciąg x_i. Chcemy dla każdego k obliczyć sumę postaci $\sum_{i=1}^{n} \max(y_i - |i - k| \cdot z, 0)$. Warto pomyśleć, jak zmieni się taka suma, gdy k zwiększymy o 1. Na rysunku 2 widać dwa zbiory, w których leżą fragmenty gruntu odpowiednio wchodzące do piramidy i wychodzące z piramidy w trakcie jej przesunięcia. Nazwijmy te zbiory *prawą* i *lewą skośną*, a ich rozmiary oznaczmy przez p_i i l_i. Jasne jest, że znając liczbę pól każdej skośnej oraz zawartość piramidy dla $k = 1$, łatwo wyznaczymy odpowiedzi dla każdego k. Jako że pierwszą piramidę możemy zbadać w czasie $O(n)$, skupimy się teraz na obliczeniu ciągu p_i w czasie $O(n)$ (wówczas ciąg l_i można będzie wyznaczyć w ten sam sposób).



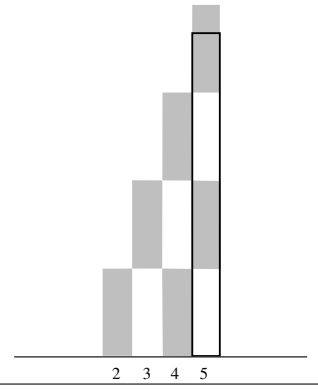
Rys. 2: Piramida, przesuwając się o jednostkę w prawo, wchłania prawą skośną i pozostawia po sobie lewą skośną.

Rozważmy pewne $j \in \{1, \ldots, n\}$. Zastanówmy się, dla jakich i, i-ta prawa skośna zawiera piasek z pozycji j. Na pewno $y_j \bmod z$ najwyższych warstw trafi do skośnej o numerze $j - \left\lfloor \frac{y_j}{z} \right\rfloor$, o ile, rzecz jasna, warstwa o takim numerze istnieje. Natomiast wszystkie skośne z przedziału $[\max(j + 1 - \left\lfloor \frac{y_j}{z} \right\rfloor, 1), j]$ dostaną po z warstw piasku, patrz także rys. 3.

Oczywiście, bezpośrednia aktualizacja skośnych doprowadziłaby nas z powrotem do czasu kwadratowego. Zamiast tego możemy jedynie zaznaczyć, gdzie zaczyna się i kończy przedział prawych skośnych, których rozmiary chcemy zwiększyć o z, a faktyczne sumowanie wykonać na samym końcu. Mówiąc dokładniej, jeśli w pomocniczej tablicy z licznikami $t[]$ na początku interesującego nas przedziału ustawimy 1, a tuż za jego końcem -1, to liczba przedziałów pokrywających skośną o numerze i będzie równa $t[1] + \ldots + t[i]$. Pozwala to wyznaczyć ciąg p_i następującym algorytmem (zakładamy, że tablice $t[]$ i $p[]$ są na początku wypełnione zerami):

```
1: for j := 1 to n do begin
2:     t[max(j + 1 − y[j] div z, 1)] += 1;
3:     t[j + 1] −= 1;
4:     if j − y[j] div z > 0 then
```

```
5:      p[j − y[j] div z] += y[j] mod z;
6:  end
7:  sum := 0;
8:  for i := 1 to n do begin
9:    sum += t[i];
10:   p[i] += z · sum;
11: end
```



Rys. 3: Niech $j = 5$, $z = 3$ i $y_5 = 11$. Rysunek przedstawia, ile piasku z piątej pozycji trafi do poszczególnych prawych skośnych: 2 warstwy zasilą p_2, zaś do p_3, p_4, p_5 trafią po 3 warstwy.

Możemy w tym momencie podsumować nasze rozważania, prezentując pseudokod funkcji rozstrzygającej, czy można dokopać się do wody przy nachyleniu nieprzekraczającym z. Funkcja zwraca najmniejszy poprawny indeks k, jeśli bezpieczny wykop jest możliwy, a w przeciwnym razie -1.

```
1:  function sprawdź_nachylenie(z, m)
2:  begin
3:    y[] := wygładź(x[], z);
4:    p[] := oblicz_prawe_skośne(y[], z);
5:    l[] := oblicz_lewe_skośne(y[], z);
6:    koszt := 0;   piramida := 0;
7:    for i := 1 to n do begin
8:      koszt += x[i] − y[i];
9:      piramida += max(y[i] − (i − 1) · z, 0);
10:   end
11:   if koszt + piramida ⩽ m then
12:     return 1;
13:   for k := 2 to n do begin
14:     piramida += p[k];
15:     piramida −= l[k − 1];
16:     if koszt + piramida ⩽ m then
17:       return k;
```

```
    end
    return −1;
end
```

Dołączając do powyższego pseudokodu wyszukiwanie binarne wartości z, uzyskujemy algorytm wykonujący $O(n \log(\max x_i))$ operacji i wykorzystujący liniową pamięć.

Rozwiązania niepoprawne

Autorzy testów przewidzieli takie błędy, jak wyrównywanie jedynie stoku wokół miejsca wykopu, pominięcie przypadku $z = 0$, czy też korzystanie ze zbyt małego typu do reprezentacji liczb całkowitych.

Inne niepoprawne rozwiązanie polega na próbie dokopania się do wody jedynie w miejscach o najmniejszej grubości piasku. Modyfikacja tej strategii, która wybiera np. 30 punktów o najmniejszej wysokości, działa nieco lepiej — jakkolwiek łatwo wskazać kontrprzykład na nią, trzeba przyznać, że nieźle radzi sobie z testami losowymi.

Rozwiązania wolniejsze

Oprócz wspomnianego na początku algorytmu, działającego w czasie $O(n^2 \log(\max x_i))$, można wymyślić szereg rozwiązań o złożoności czasowej $O(n \log n \log(\max x_i))$. Jednym z nich jest modyfikacja rozwiązania wzorcowego, korzystająca z tzw. drzewa licznikowego, zwanego też drzewem przedziałowym[1], do obliczenia zawartości skośnych. Innym przykładem jest zastosowanie wolniejszego algorytmu wygładzania terenu, w którym znajdujemy kolejno miejsca o najniższej wysokości gruntu, używając do tego kolejki priorytetowej. Wiadomo, że z najniższego miejsca nie warto usuwać piasku, więc można od razu zaktualizować sąsiednie pozycje i wyrzucić minimum z kolejki. Widać, że po wykonaniu tej operacji n razy teren zostanie wygładzony. Rozwiązania o takiej złożoności mogły zdobyć 60 punktów lub więcej, w zależności od jakości implementacji.

Rozwiązania działające liniowo względem m lub $\max x_i$ nie miały większych szans na osiągnięcie wyniku powyżej 30 punktów. Należały do nich w szczególności programy niekorzystające z wyszukiwania binarnego.

Testy

Do tworzenia testów używane były funkcje generujące losowe ciągi liczb: rosnące, malejące i dowolne. W poniższym opisie „dołek" oznacza ciąg liczb, który do pewnego miejsca jest malejący, a potem rosnący. Natomiast „nierówny dołek" powstaje z dołka przez podzielenie go na fragmenty równej długości i losowe poprzestawianie elementów w każdym z tych fragmentów z osobna. Podobnie definiujemy „górkę" i „nierówną górkę". Tabelka na następnej stronie zawiera krótkie opisy testów z zestawu.

[1] Opis tej struktury danych można znaleźć np. w opracowaniu zadania *Tetris 3D* z XIII Olimpiady Informatycznej [13] czy opracowaniu zadania *Koleje* z IX Olimpiady Informatycznej [9].

Nazwa	n	m	Opis
stu1.in	71	236	mały dołek z nierównościami na środku
stu2a.in	237	821	mały dołek z nierównościami na środku otoczony spadami
stu2b.in	842	681 835	niewielki losowy test, odpowiedź 0
stu3.in	2 042	4 423	niewielki nierówny dołek
stu4a.in	5 000	2 521	trzy niewielkie dołki, skrajne są nierówne
stu4b.in	10 000	5	specyficzny test z małymi liczbami, prócz jednej na środku
stu4c.in	10 000	10^{12}	duża liczba dołków, z których jeden jest szerszy i płytszy
stu5a.in	10 000	500 000	średni test, dołek otoczony losowymi wartościami
stu5b.in	10 000	$\approx 10^{14}$	dosyć duży losowy test, odpowiedź 0
stu5c.in	10 000	10^{11}	duża liczba losowych odcinków, z których jeden jest szerszy i średnio niższy
stu6a.in	486 000	2 414 746 423	54 średnie dołki
stu6b.in	75 000	811 178 223	dołek z nierównościami na środku
stu7a.in	100 000	828	specyficzny test z jedną poprawną odpowiedzią
stu7b.in	140 000	411 651 544	dołek z nierównościami na środku
stu7c.in	450 000	898 889	duży losowy test
stu8a.in	150 000	2	specyficzny test z jedną poprawną odpowiedzią
stu8b.in	90 000	1 059 865 233	dołek z nierównościami na środku
stu9a.in	400 000	352 247	duży dołek z nierównościami na środku
stu9b.in	550 000	$\approx 3 \cdot 10^{14}$	duży dołek z nierównościami na środku
stu10a.in	500 000	10^{18}	duża nierówna górka, odpowiedź 0
stu10b.in	700 000	$\approx 10^{12}$	duży dołek z nierównościami na środku
stu11a.in	1 000 000	15 486 352 148	150 nierównych dołków
stu11b.in	1 000 000	500 000 000	maksymalny dołek
stu11c.in	1 000 000	$\approx 5 \cdot 10^{12}$	duży dołek z nierównościami na środku
stu12a.in	1 000 000	$\approx 5 \cdot 10^{11}$	maksymalny dołek z nierównościami na środku
stu12b.in	750 000	2	specyficzny test z jedną poprawną odpowiedzią

Bibliografia

[1] *I Olimpiada Informatyczna 1993/1994*. Warszawa–Wrocław, 1994.

[2] *II Olimpiada Informatyczna 1994/1995*. Warszawa–Wrocław, 1995.

[3] *III Olimpiada Informatyczna 1995/1996*. Warszawa–Wrocław, 1996.

[4] *IV Olimpiada Informatyczna 1996/1997*. Warszawa, 1997.

[5] *V Olimpiada Informatyczna 1997/1998*. Warszawa, 1998.

[6] *VI Olimpiada Informatyczna 1998/1999*. Warszawa, 1999.

[7] *VII Olimpiada Informatyczna 1999/2000*. Warszawa, 2000.

[8] *VIII Olimpiada Informatyczna 2000/2001*. Warszawa, 2001.

[9] *IX Olimpiada Informatyczna 2001/2002*. Warszawa, 2002.

[10] *X Olimpiada Informatyczna 2002/2003*. Warszawa, 2003.

[11] *XI Olimpiada Informatyczna 2003/2004*. Warszawa, 2004.

[12] *XII Olimpiada Informatyczna 2004/2005*. Warszawa, 2005.

[13] *XIII Olimpiada Informatyczna 2005/2006*. Warszawa, 2006.

[14] *XIV Olimpiada Informatyczna 2006/2007*. Warszawa, 2007.

[15] *XV Olimpiada Informatyczna 2007/2008*. Warszawa, 2008.

[16] *XVI Olimpiada Informatyczna 2008/2009*. Warszawa, 2009.

[17] *XVII Olimpiada Informatyczna 2009/2010*. Warszawa, 2010.

[18] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Projektowanie i analiza algorytmów komputerowych*. PWN, Warszawa, 1983.

[19] L. Banachowski, A. Kreczmar. *Elementy analizy algorytmów*. WNT, Warszawa, 1982.

[20] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych*. WNT, Warszawa, 1996.

[21] J. Bentley. *Perełki oprogramowania.* WNT, Warszawa, 1992.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Wprowadzenie do algorytmów.* WNT, Warszawa, 2004.

[23] M. de Berg, M. van Kreveld, M. Overmars. *Geometria obliczeniowa. Algorytmy i zastosowania.* WNT, Warszawa, 2007.

[24] R. L. Graham, D. E. Knuth, O. Patashnik. *Matematyka konkretna.* PWN, Warszawa, 1996.

[25] D. Harel. *Algorytmika. Rzecz o istocie informatyki.* WNT, Warszawa, 1992.

[26] D. E. Knuth. *Sztuka programowania.* WNT, Warszawa, 2002.

[27] W. Lipski. *Kombinatoryka dla programistów.* WNT, Warszawa, 2004.

[28] E. M. Reingold, J. Nievergelt, N. Deo. *Algorytmy kombinatoryczne.* WNT, Warszawa, 1985.

[29] K. A. Ross, C. R. B. Wright. *Matematyka dyskretna.* PWN, Warszawa, 1996.

[30] R. Sedgewick. *Algorytmy w C++. Grafy.* RM, 2003.

[31] S. S. Skiena, M. A. Revilla. *Wyzwania programistyczne.* WSiP, Warszawa, 2004.

[32] P. Stańczyk. *Algorytmika praktyczna. Nie tylko dla mistrzów.* PWN, Warszawa, 2009.

[33] M. M. Sysło. *Algorytmy.* WSiP, Warszawa, 1998.

[34] M. M. Sysło. *Piramidy, szyszki i inne konstrukcje algorytmiczne.* WSiP, Warszawa, 1998.

[35] R. J. Wilson. *Wprowadzenie do teorii grafów.* PWN, Warszawa, 1998.

[36] N. Wirth. *Algorytmy + struktury danych = programy.* WNT, Warszawa, 1999.

[37] M. Adamaszek. Zliczamy inwersje. *Delta*, czerwiec, 2008.