Zawody I stopnia

opracowania zadaĆń

Wojciech Rytter
treść zadania, opracowanie

Grzegorz Jakacki
program wzorcowy

Liczba wyborów symetrycznych

Dane są dwa ciągi słów (x_i) *oraz* (y_i)*, gdzie:* $1 \leqslant i \leqslant n$ *oraz* $1 \leqslant n \leqslant 30$*. Kolejno dla każdego i od 1 do n wybieramy jedno z dwóch słów:* x_i *albo* y_i *i wybrane słowa składamy (kolejne wybrane słowo dopisujemy do poprzednich z prawej strony).*
Wybór *składa się z n decyzji, czy w kolejnym kroku wybrać odpowiedni i-ty wyraz z pierwszego, czy z drugiego ciągu słów. Bardziej formalnie: wybór jest n wyrazowym ciągiem liczb 1 lub 2.*
Różne wybory mogą dać w wyniku to samo słowo.
Wybór nazywamy **symetrycznym***, gdy jego wynikiem jest palindrom — tzn. takie słowo, które nie zmienia się, gdy je czytamy od strony prawej do lewej.*

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego LIC.IN liczbę n i dwa n-wyrazowe ciągi słów* (x_i) *i* (y_i)*,*
- *oblicza liczbę wszystkich wyborów symetrycznych dla tych dwóch ciągów słów,*
- *zapisuje wynik w pliku tekstowym LIC.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego LIC.IN jest zapisana jedna liczba całkowita dodatnia $n \leqslant 30$*.*
W następnych n wierszach są zapisane kolejne słowa ciągu (x_i) *— każde w osobnym wierszu.*
W następnych n wierszach są zapisane — w taki sam sposób — kolejne słowa ciągu (y_i)*.*
Wszystkie słowa składają się wyłącznie z małych liter alfabetu angielskiego od a do z i mają długość nie mniejszą niż 1 i nie większą niż 400.

Wyjście

W pliku tekstowym LIC.OUT należy zapisać jedną liczbę całkowitą nieujemną, a mianowicie liczbę wszystkich wyborów symetrycznych.

Przykład

Dla pliku LIC.IN:

```
5
ab
a
a
ab
a
a
baaaa
a
a
ba
```

poprawnym rozwiązaniem jest plik LIC.OUT:

```
12
```

Twój program powinien szukać pliku LIC.IN w katalogu bieżącym i tworzyć plik LIC.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę LIC.???, gdzie *zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku LIC.EXE*

UWAGA DO TREŚCI ZADANIA

W treści zadania znalazła się pewna niedokładność. Ostatnie zdanie punktu „Wejście" miałoby sens bliższy zamierzonemu przez autora zadania, gdyby brzmiało:

> *Wszystkie słowa składają się wyłącznie z małych liter alfabetu angielskiego od a do z i mają łączną długość nie mniejszą niż 1 i nie większą niż 400.*

ROZWIĄZANIE

Zadanie jest dosyć proste, łatwo rozwiązuje się je przez sprowadzenie do problemu obliczenia liczby wszystkich ścieżek prowadzących od wierzchołka początkowego **start** do wierzchołka końcowego **koniec** w pewnym grafie acyklicznym G. Jak się okaże, bezpośrednia konstrukcja grafu G w programie nie jest konieczna.

Rozwiązywanie zadania w sposób „naiwny" przez zbadanie wszystkich możliwych wyborów prowadzi do programu działającego w przypadku pesymistycznym zbyt długo, aby można było taki program zaakceptować (liczba możliwych wyborów jest wykładnicza).

UWAGA: Jeśli zastąpimy liczenie wszystkich wyborów przez liczenie wszystkich różnych słów symetrycznych, które możemy otrzymać, to zadanie staje się dużo trudniejsze. Nie jest znany żaden wielomianowy algorytm dla takiej wersji zadania.

Na wejściu dostajemy ciąg słów $x_1, \ldots, x_n$, $y_1, \ldots, y_n$. Załóżmy, że N jest sumą długości wszystkich słów. Słowa pamiętamy w tablicy T, jako jedno długie słowo $x_1y_1x_2y_2\ldots x_ny_n$. Dodatkowo przechowujemy informacje o tym, gdzie w tablicy T są końce i początki słów x_i, y_i.

Chociaż nasze zadanie ma charakter tekstowy, to sprowadzimy je do problemu teoriografowego.

Wyobraźmy sobie, że mamy dwa wskaźniki przesuwające się po tablicy T. Jeden z nich „wędruje" od strony lewej do prawej, a drugi od prawej do lewej. Wskaźniki służą do wybierania po jednym słowie z każdej pary i przesuwają się w sposób zsynchronizowany, tak że tyle samo liter jest pobieranych zarówno z lewej, jak i z prawej strony. Przesuwając wskaźniki sprawdzamy jednocześnie symetrię tekstu. W momencie, gdy wskaźniki się spotkają wiemy, że znaleźliśmy słowo symetryczne i dokonaliśmy wyboru z każdej pary x_k, y_k. Wskaźniki przesuwamy tylko wtedy, gdy kolejne litery są identyczne (rozszerzamy symetrię od zewnątrz do środka).

Rozważmy skierowany acykliczny graf $G = (V, E)$, którego budowę omówimy na przykładzie z treści zadania. W tym celu posłużymy się rys. 1. Zbiór V składa się ze wszystkich par (i, j), gdzie $1 \leqslant i < j \leqslant N$, zwanych **wierzchołkami wewnętrznymi**, oraz dodatkowych wierzchołków *start* i *koniec*.

Każdy wewnętrzny wierzchołek jest parą (i, j) interpretowaną jako

$$(\text{pozycja lewego wskaźnika}, \text{pozycja prawego wskaźnika})$$

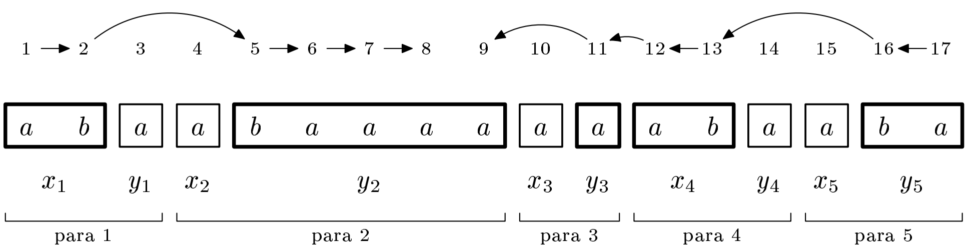
Tablica T zawiera słowo $ab|a|a|baaaa|a|a|ab|a|a|ba$, gdzie symbol '|' jedynie pokazuje granice między poszczególnymi słowami, ale w rzeczywistości nie ma go w tablicy. Zatem tablica ta ma długość $N = 17$.

Lewy wskaźnik przesuwa się zgodnie ze strzałkami $\rightarrow$ i $\curvearrowright$, a prawy zgodnie z $\leftarrow$ i $\curvearrowleft$. W grafie G mamy krawędź $(1, 17) \rightsquigarrow (2, 16)$ gdyż 2-gi i 16-ty symbol są identyczne, a przy tym jesteśmy wewnątrz pojedynczych słów. Natomiast krawędź $(2, 16) \rightsquigarrow (5, 13)$ bierze się stąd, że podobnie jak poprzednio, odpowiednie symbole są identyczne oraz pozycje 2 i 16 są końcami pojedynczych słów (czytając od lewej do prawej — pozycja 2 — i od prawej do lewej — pozycja 16), natomiast pozycje 5 i 13 są początkami pojedynczych słów z następnych par w kolejności.

Łatwo zauważyć następujący fakt:

Fakt 1: Liczba wyborów symetrycznych jest równa liczbie wszystkich ścieżek prowadzących od wierzchołka *start* do wierzchołka *koniec* w grafie acyklicznym G.

Rys. 1 Załóżmy, że z każdej pary słów wybieramy słowo w pogrubionej ramce. Jest to wybór symetryczny. Odpowiada to przykładowej ścieżce w grafie G przedstawionej pod rysunkiem.



$$\textit{start} \rightsquigarrow (1,17) \rightsquigarrow (2,16) \rightsquigarrow (5,13) \rightsquigarrow (6,12) \rightsquigarrow (7,11) \rightsquigarrow (8,9) \rightsquigarrow \textit{koniec}$$

Liczbę ścieżek liczymy następująco. Przechodzimy graf w porządku topologicznym, to znaczy takim porządku, że dla każdego wierzchołka wszystkie jego poprzedniki są odwiedzane wcześniej niż on sam. Liczymy liczbę ścieżek od wierzchołka *start* do każdego wierzchołka v, jako sumę liczb ścieżek dla jego bezpośrednich poprzedników. Wierzchołek $v = \textit{start}$ ma jedną ścieżkę od siebie samego do siebie samego.

W programie oprócz tablicy T mamy tablice *nr_slowa* i *skok*, które służą do wskazywania początków i końców pojedynczych słów x_k, y_k w tablicy T oraz pomagają wyznaczać krawędzie grafu G. Sam graf jest przechowywany tylko pośrednio — dla każdego wierzchołka, do którego istnieje ścieżka z wierzchołka *start* pamięta się obliczaną przez program liczbę wszystkich takich ścieżek oraz liczbę poprzedników wierzchołka (wykorzystywaną przy przechodzeniu grafu w porządku topologicznym). Istotnym problemem technicznym jest duży rozmiar tablicy, w której przechowuje się powyższe wartości (tablica G w programie). Wykorzystując fakt, że jest to tablica trójkątna (używamy tylko komórek po jednej stronie przekątnej tablicy), można zmniejszyć rozmiar potrzebnej pamięci. Nie konstruuje się krawędzi grafu. Fakt istnienia krawędzi każdorazowo wyznacza się na podstawie zawartości tablic T, *nr_slowa* i *skok*.

TESTY

Do sprawdzania rozwiązań zawodników użyto 12-tu testów LIC0.IN–LIC11.IN.

- LIC0.IN — test z treści zadania;
- LIC1.IN–LIC8.IN — testy sprawdzające poprawność przyjętej metody rozwiązania;
- LIC9.IN–LIC11.IN — testy sprawnościowe. Ich celem było sprawdzenie szybkości działania algorytmu.

Bogdan S. Chlebus
treść zadania, opracowanie

Krzysztof Sobusiak
program wzorcowy

Skok w bok

Plansza do gry w „Skok w bok” jest nieskończoną taśmą pól, nieograniczoną zarówno w lewo jak i w prawo. Na polach planszy stoją pionki. Ich liczba jest skończona. Na jednym polu może stać wiele pionków jednocześnie. Zakładamy, że pierwsze od lewej pole, na którym jest co najmniej jeden pionek, ma numer 0. Pola na prawo od niego są oznaczone kolejno liczbami naturalnymi 1, 2, 3 itd., a pola w lewo liczbami ujemnymi: -1, -2, -3 itd. Ustawienie pionków na taśmie, które będziemy także nazywać **konfiguracją**, *można opisać w ten sposób, że dla każdego pola, na którym jest co najmniej jeden pionek, podaje się numer pola i liczbę pionków na tym polu.*
Są dwa rodzaje ruchów zmieniających konfigurację: skok w prawo i skok w lewo.
Skok w prawo *polega na zabraniu po jednym pionku z wybranych dwóch sąsiednich pól o numerach p i $p+1$ i dodaniu jednego pionka na polu $p+2$.*
Skok w lewo: *zabieramy jeden pionek z pola $p+2$, a dodajemy po jednym na polach p i $p+1$.*
Mówimy, że konfiguracja jest **końcowa,** *jeśli na dowolnych dwóch sąsiednich polach znajduje się co najwyżej jeden pionek.*
Dla każdej konfiguracji istnieje dokładnie jedna konfiguracja końcowa, którą można z niej otrzymać w wyniku skończonej liczby ruchów w prawo lub w lewo.

Zadanie

Ułóż program, który:

- *wczytuje opis konfiguracji początkowej z pliku tekstowego SKO.IN;*
- *znajduje konfigurację końcową, do jakiej można doprowadzić daną konfigurację początkową i zapisuje wynik w pliku tekstowym SKO.OUT.*

Wejście

W pierwszym wierszu pliku SKO.IN jest zapisana jedna liczba całkowita dodatnia n. Jest to liczba niepustych pól danej konfiguracji początkowej. $1 \leqslant n \leqslant 10000$ (dziesięć tysięcy).
W każdym z kolejnych n wierszy znajduje się opis jednego niepustego pola konfiguracji początkowej w postaci pary liczb całkowitych oddzielonych odstępem. Pierwsza liczba to numer pola, a druga — to liczba pionków na tym polu. Te opisy są uporządkowane rosnąco względem numerów pól. Największy numer pola nie przekracza 10000 (dziesięć tysięcy), a liczba pionków na żadnym polu nie przekracza 10^8 (sto milionów).

Wyjście

W pierwszym wierszu pliku tekstowego SKO.OUT należy zapisać konfigurację końcową, do której można przekształcić daną konfigurację początkową — numery niepustych pól konfiguracji końcowej. Numery te powinne być uporządkowane rosnąco. Liczby w wierszu powinny być pooddzielane pojedynczym odstępem.

Przykład

Dla pliku tekstowego SKO.IN:

```
2
0 5
3 3
```

poprawnym rozwiązaniem jest plik SKO.OUT:

```
-4 -1 1 3 5
```

ROZWIĄZANIE

Opis rozwiązania zadania rozpoczniemy od kilku prostych uwag. Chociaż ruchy pionków w grze są opisane w terminach pojedynczych pionków (dla każdego pola gdzie występują zmiany liczby pionków), to efektywne rozwiązanie wymaga symulowania ruchów wieloma pionkami jednocześnie.

Zauważmy, że jeżeli na każdym niepustym polu leży co najwyżej jeden pionek, to taką konfigurację można przekształcić do końcowej wykonując tylko ruchy w prawo (proste ćwiczenie). Jeżeli natomiast wszystkie pionki leżą na jednym polu, to ruch w lewo jest konieczny. Stąd wynika, że algorytm musi przeplatać ciągi ruchów w prawo z ciągami ruchów w lewo.

Spodziewamy się, że zmiany konfiguracji prowadzące do zmniejszania się całkowitej liczby pionków mogą prowadzić do konfiguracji końcowych lub im bliskich. Ruch w prawo zmniejsza liczbę pionków, natomiast ruch w lewo zwiększa. Przed każdym wykonaniem ruchu w lewo musimy więc wykazać ostrożność, aby nie spowodować nadmiernego wzrostu liczby pionków, nawet przy lokalnym uproszczeniu konfiguracji, które w ogólnym rozliczeniu okazać się może pozorne.

JEDNOZNACZNOŚĆ KONFIGURACJI KOŃCOWEJ

Przypominamy pojęcie **liczb Fibonacciego** — tworzą one ciąg nieujemnych liczb całkowitych $\langle F_i \rangle$, dla $i \geq 0$, określony rekurencyjnie w następujący sposób:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_{i+2} = F_i + F_{i+1}$$

Określamy **wyrażenia** jako skończone sumy liczb Fibonacciego, zakładając, że te liczby zapisane są zawsze w kolejności niemalejących wskaźników. Rozważamy następujące operacje na takich wyrażeniach:

- F_{i+2} zastępujemy przez sumę $F_i + F_{i+1}$;
- sumę $F_i + F_{i+1}$ zastępujemy przez F_{i+2}.

Takie przekształcanie wyrażeń nie zmienia ich wartości.

Na związek między sumami liczb Fibonacciego a konfiguracjami pionków na planszy wskazuje naturalne podobieństwo między operacjami wykonania skoku w lewo lub w prawo, a operacjami wymiany podwyrażeń $F_i + F_{i+1}$ i F_{i+2}. Jeżeli od jednego wyrażenia można przejść do drugiego przez wykonanie skończonego ciągu takich operacji, to nazwiemy je **podobnymi**. Jeżeli od jednej konfiguracji można przejść do innej przez wykonanie skończonego ciągu ruchów, to także takie konfiguracje nazwiemy **podobnymi**. Chcielibyśmy konfiguracjom przypisać wyrażenia w taki sposób, żeby podobne konfiguracje odpowiadały podobnym wyrażeniom. Tu napotykamy na pewne trudności. Konfiguracje są jednoznacznie określone przez względne położenie pionków na planszy. Pola planszy są ponumerowane tylko w celu wygodnego opisu konfiguracji, natomiast liczby Fibonacciego są określone przez swoje wskaźniki. Innymi słowy: gdy konfigurację przesuniemy o jedno pole w prawo to będzie ona „taka sama”, natomiast gdy w sumie liczb Fibonacciego zwiększymy każdy wskaźnik, to dostaniemy większą liczbę (poza przypadkiem wyrazeń F_1 i F_2). W opisie konfiguracji mogą pojawić się pola o numerach ujemnych, natomiast liczby Fibonacciego mają wskaźniki nieujemne. Największą trudnością logiczną jest jednak to, że wykonując odpowiednio wiele ruchów w lewo możemy utworzyć dowolnie wiele pionków na planszy!

Aby uniknąć takich trudności, najprościej rozważyć **grę pomocniczą**, bardzo podobną do właściwej gry „Skok w bok”. Przyjmijmy, że plansza jest ciągiem pól nieskończonym tylko w jedną, prawą stronę. Pola są ponumerowane kolejnymi liczbami całkowitymi dodatnimi, począwszy od skrajnie lewego pola, które ma numer 1. Ruchy w prawo można wykonywać z każdego pola, natomiast ruchy w lewo tylko z pól o numerach nie mniejszych niż 3. Dodatkowo wprowadzamy regułę, że pionki między polami 1 oraz 2 można dowolnie przesuwać. Konfiguracji przyporządkowujemy liczbę $\sum_{i\geqslant 1} a_i \cdot F_i$, gdzie a_i jest liczbą pionków na polu i (suma jest skończona, ponieważ tylko skończenie wiele a_i jest niezerowych). Jest jasne, że takie przyporządkowanie sum liczb Fibonacciego konfiguracjom dla gry pomocniczej jest dobre w tym sensie, że konfiguracjom podobnym odpowiadają podobne wyrażenia.

Wiadomo że każdą liczbę całkowitą dodatnią n można jednoznacznie przedstawić w postaci sumy $\sum_{i>1} b_i \cdot F_i$, gdzie $b_i = 0$ poza skończoną liczbą indeksów i, oraz jeżeli $b_i \neq 0$, to $b_i = 1$ i $b_{i-1} = b_{i+1} = 0$ (patrz [13]). Takie sumy to wyrażenia, w których każda liczba Fibonacciego F_i, dla $i > 0$, występuje co najwyżej jeden raz i dwie liczby o sąsiednich indeksach nie występują jednocześnie. Stąd od razu wynika, że dla każdej konfiguracji istnieje dokładnie jedna podobna do niej konfiguracja końcowa w grze pomocniczej.

Jednoznaczność końcowej konfiguracji dla gry „Skok w bok” także zachodzi. Można to pokazać, odwołując się do powyższej własności liczb Fibonacciego (o jednoznaczności przedstawienia liczb naturalnych w postaci odpowiedniego wyrażenia). Szczegóły pozostawiamy jako ćwiczenie. Jako wskazówkę proponujemy takie ponumerowanie pól planszy aby pierwsze od lewej niepuste pole miało numer o tak dużej wartości, żeby w konfiguracji końcowej występowały tylko pola o dodatnich numerach.

ROZŁADOWYWANIE

Pole z co najmniej trzema pionkami nazywamy **wieżą**. Ciąg ruchów, który zmniejsza liczbę pionków danej wieży, jednocześnie zmniejszając łączną liczbę pionków na planszy, nazywamy **rozładowaniem wieży**. Poniżej opisujemy jeden z takich sposobów.

Notacja

$$\ldots, a_1, a_2, \ldots, a_n, \ldots$$

oznacza konfigurację, w której na i-tym polu spośród n kolejnych, znajduje się a_i pionków. Wykonanie ruchu i przejście do następnej konfiguracji zapisujemy jako dwie konfiguracje przedzielone strzałką. Na przykład ruch w prawo możemy zapisać jako

$$\ldots, x+1, y+1, z \ldots \quad \Longrightarrow \quad \ldots, x, y, z+1, \ldots$$

Natomiast ruch w lewo zapisujemy jako

$$\ldots, x, y, z+1 \ldots \quad \Longrightarrow \quad \ldots, x+1, y+1, z, \ldots$$

Rozważmy następujący ciąg trzech kolejnych ruchów:

$$\begin{aligned} \ldots, a, b, c+3, d, e, \ldots \quad &\Longrightarrow \quad \ldots, a+1, b+1, c+2, d, e \ldots \quad \Longrightarrow \\ &\Longrightarrow \quad \ldots, a+1, b, c+1, d+1, e \ldots \quad \Longrightarrow \\ &\Longrightarrow \quad \ldots, a+1, b, c, d, e+1 \ldots \end{aligned}$$

Nazwiemy go **pojedynczym rozładowaniem**. Taka operacja usuwa trzy pionki z pola i umieszcza dwa na innych polach. Jeżeli $3k$ jest maksymalną wielokrotnością 3 nie większą niż liczba pionków na danym polu, to możemy zastosować jednocześnie operację pojedynczego rozładowania do każdej z k trójek, co nazwiemy też **rozładowaniem**. Taka operacja usuwa $3k$ pionków z rozważanego pola i umieszcza po k pionków na dwóch innych polach (razem $2k$).

ALGORYTM OBLICZANIA KONFIGURACJI KOŃCOWEJ

Konfiguracja reprezentowana jest jako skończona lista kolejnych pól. Możemy ograniczyć się tylko do niepustych pól, ale to zależy od szczegółów implementacji. Z polem wiążemy informację o liczbie pionków. Algorytm obliczania konfiguracji końcowej składa się z dwóch faz.

Faza 1

while istnieje pole zawierające co najmniej 3 pionki **do**

przejrzyj listę niepustych pól w kolejności od lewej do prawej i rozładuj
każde zawierające co najmniej 3 pionki

Po każdym rozładowaniu cofamy się o dwa pola, żeby sprawdzić czy nie pojawią się tam co najmniej trzy pionki, ale takich pól nie rozładowujemy w tej iteracji pętli.

Niech *pion* będzie funkcją zwracającą liczbę pionków w polu.

Faza 2

pole := numer skrajnie prawego niepustego pola;
pole := *pole* − 1;
while konfiguracja nie jest końcowa **do**
case
  $pion(pole) > 0$ i $pion(pole+1) > 0$:
    wykonaj ruch w prawo z *pole*;
    pole := *pole* + 2;
  $pion(pole) = 2$ i $pion(pole-1) > 0$:
    pole := *pole* − 1;
  $pion(pole) = 2$ i $pion(pole+1) = 0$:
    wykonaj ruch w lewo z *pole*;
    wykonaj ruch w prawo z *pole* − 1;
    pole := *pole* + 1;
  $pion(pole) = 3$:
    rozładuj *pole*;
    pole := *pole* + 2;
  pole jest numerem skrajnie lewego niepustego pola:
    zakończ fazę 2.;
  w pozostałych przypadkach:
    pole := numer następnego niepustego pola z lewej;

Przyjmujemy, że warunki w konstrukcji **case** są sprawdzane po kolei, a po znalezieniu pierwszego spełnionego warunku wykonujemy stojącą przy nim instrukcję (i na tym wykonanie **case** się kończy).

POPRAWNOŚĆ ALGORYTMU

Faza 1. kiedyś się zakończy, bowiem każde rozładowanie pola zmniejsza łączną liczbę pionków na planszy. Faza 2. zaczyna się w sytuacji gdy na każdym polu leżą co najwyżej dwa pionki. Idziemy od strony prawej do lewej starając się, by na prawo była już sytuacja taka, jak w konfiguracji końcowej. Poprawność fazy 2 wynika z zachowania następujących niezmienników po każdej iteracji instrukcji **case**:

- co najwyżej jedno pole zawiera trzy pionki;
- na prawo od *pole* nie ma pola zawierającego więcej niż dwa pionki;
- jeżeli *pole* i *pole* + 1 są niepuste, to na prawo od *pole* nie ma już innych takich par sąsiednich pól.

Formalny dowód przebiega przez indukcję i polega na rozważeniu wszystkich możliwych przypadków.

Niezmienniki z instrukcji **case** pokazują, że nie nastąpi „spiętrzenie” pionków. Jednocześnie zauważmy, że w każdych czterech kolejnych iteracjach instrukcji **case** albo następuje zmniejszenie łącznej liczby pionków albo *pole* przyjmuje wartość mniejszą od wszystkich dotychczasowych.

IMPLEMENTACJA ALGORYTMU

Listę pól pamiętamy w tablicy liczb całkowitych; elementami tablicy są liczby pionków stojących na polach. Tablica ma rozmiar taki, jak podany rozmiar planszy, powiększony z obu końców tak, by pomieścić pola potrzebne do trzymania wież powstałych przy rozładowywaniu wysokich wież znajdujących sie na skraju planszy w początkowej konfiguracji. Rozmiar powiększenia wyraża się logarytmem z maksymalnej dopuszczalnej na wejściu wysokości wieży pionków, ponieważ każde rozładowanie zmniejsza wysokość rozładowywanej wieży o jedną trzecią. W fazie 1 wieże, które pozostają jeszcze do rozładowania trzymamy w kolejce, implementowanej w tablicy w taki sposób, że wieże są pobierane w kolejności ich wstawiania. Program realizujący tę implementację znajduje się w pliku SKO.PAS.

INNE ROZWIĄZANIA

Można rozważać inne rozwiązania. Omówimy algorytm oparty na następującej zasadzie: wykonujemy ruchy w prawo, dopóki to jest możliwe, a gdy nie jest, to wykonujemy ruch w lewo. W takich sytuacjach ruch oznacza skok wieloma pionkami jednocześnie. Skok w prawo najlepiej wykonać maksymalną możliwą liczbą pionków jednocześnie. Przy skoku w lewo dobrze jest tak rozłożyć pionki, aby zapewnić sobie potem długi ciąg ruchów w prawo. Można to osiagnąć stosują zasadę „złotego podziału”. Niech $\phi = (1+\sqrt{5})/2$. Przypuśćmy, że mamy konfigurację

$$\ldots, x, 0, w, 0 \ldots$$

którą nazywamy **izolowaną wieżą**, o ile $w > 1$. Chcielibyśmy wykonać ruch w lewo:

$$\ldots, x, 0, w, 0 \ldots \implies \ldots, x + a \cdot w, a \cdot w, (1-a) \cdot w, 0, \ldots$$

a następnie w prawo:

$$\ldots, a \cdot w, a \cdot w, (1-a) \cdot w, 0, \ldots \implies \ldots, a \cdot w, 0, (1-2a) \cdot w, a \cdot w, \ldots$$

i kontynuować ruchy w prawo. Najdłuższy możliwy ciąg ruchów w prawo otrzymamy, gdy iloraz wysokości dwóch sąsiednich wież zmienia się jak najmniej, to znaczy gdy dwa ułamki

$$\frac{a}{1-a} \quad \text{oraz} \quad \frac{1-2a}{a}$$

są jak najbliższe co do wartości. Rozwiązując równanie otrzymane przez przyrównanie do siebie ułamków dostajemy, że $a = 1 + \phi$. Szybka implementacja takiego algorytmu wygląda następująco. Zaczynając z prawej strony przesuwamy się w lewo szukając pierwszej możliwości skoku w prawo lub izolowanej wieży. Jeżeli skok w prawo jest możliwy, wykonujemy serię takich skoków w prawo „do oporu”. Jeżeli napotkamy na izolowaną wieżę, wykonujemy ruch w lewo zgodnie z zasadą złotego podziału, po czym kontynuujemy ruchy w prawo.

Omówimy także krótko pułapki tego zadania. Typowe błędy to:

- przyjęcie, że każda konfiguracja zawsze zmieści się w tablicy pamiętającej konfigurację początkową,
- niewłaściwe przeplatanie skoków w lewo ze skokami w prawo, tak, że doprowadza to do zapętlenia.

Rozwiązania poprawne ale bardzo czasochłonne też nie są dobre. Typowy błąd to wykonywanie skoków tylko jednym pionkiem na raz, co prowadzi do algorytmu o dużej złożoności czasowej. Ciekawszy przykład nieefektywnego rozwiązania to algorytm skonstruowany podobnie do metody złotego podziału, ale rozkładający wieże na trzy prawie równe części, co przy odpowiednio dobranych danych prowadzi do bardzo złego zachowania (ćwiczenie dla czytelnika).

TESTY

Testy do tego zadania znajdują się w plikach SKO0.IN–SKO12.IN. Można je podzielić na testy poprawności i testy złożoności. Testy poprawności:

- SKO0.IN — test z treści zadania;
- SKO1.IN — prosty test wymagający jedynie skoków w prawo;
- SKO2.IN — prosty test wymagający skoków w prawo i w lewo;
- SKO3.IN — konfiguracja końcowa tożsama wejściowej — bez skoków;
- SKO4.IN — prosty test z rozwiązaniem zawierającym pola o numerach ujemnych;
- SKO5.IN — dwie sąsiadujące ze sobą wieże Fibonacciego o wysokościach F_{37} i F_{38}.

Testy złożoności:

- SKO6.IN — pojedyncza wieża wysokości 10^8;
- SKO7.IN — dwie wieże wysokości 10^8 na polach K_0 i K_{10000};
- SKO8.IN — ciąg 10000 małych wież $3, 2, 2, \ldots, 2, 2, 3$;
- SKO9.IN — test losowy: ciąg 100 wież wysokości $\leqslant 10^2$;
- SKO10.IN — test losowy: ciąg 1000 wież wysokości $\leqslant 10^4$;
- SKO11.IN — test losowy: ciąg 10000 wież wysokości $\leqslant 10^8$;
- SKO12.IN — konfiguracja rzadka: 120 wież o wysokości 10^8 w odstępach co 80 pól.

Stanisław Waligórski
treść zadania

Jacek Chrząszcz
program wzorcowy, opracowanie

Tanie podróże

Pasażerowie autokarów turystycznych, przewożących wycieczki transkontynentalną autostradą, spędzają w drodze wiele dni, zatem opłaty za noclegi stanowią poważną część kosztów podróży.
Ze względu na bezpieczeństwo jazdy i wygodę pasażerów każdy autokar jedzie tylko w ciągu dnia i nie może przejechać więcej niż 800 km dziennie. Noc na trasie (poza jej początkiem i końcem) pasażerowie i kierowca spędzają w hotelach. Dotychczas planowano przejazdy tak, by liczba noclegów na trasie była jak najmniejsza. Dążąc do obniżki kosztów przedsiębiorstwo przewozowe zdecydowało zbadać, czy opłaci się układanie planów podróży tak, by suma opłat za noclegi była możliwie najniższa, nawet gdyby to miało przedłużyć podróż. W tych obliczeniach można korzystać z ofert hoteli położonych przy autostradzie. W każdej ofercie jest podana odległość hotelu od początku trasy i cena jednego noclegu jednej osoby.
Podróż jest tylko w jedną stronę. Trasa nie ma rozgałęzień. Przez każdy punkt trasy autokar przejeżdża tylko jeden raz. Nigdzie na trasie nie ma dwóch hoteli w jednym punkcie, więc dla zidentyfikowania hotelu wystarczy podać jego odległość od początku trasy. Nie planuje się noclegu na początku ani na końcu trasy. Liczba osób w autobusie nie zmienia się i w każdym hotelu wszyscy (łącznie z kierowcą) płacą za nocleg jednakowo — zgodnie z ofertą. Pojemność hoteli jest na tyle duża, że nie istnieje problem braku miejsc. Zawsze można liczyć na to, że w dowolnej chwili w każdym hotelu będzie wystarczająco dużo wolnych miejsc, by przenocować wszystkich pasażerów autobusu. Na każdym odcinku trasy o długości 800 km jest przynajmniej jeden hotel, co oznacza, że przejechanie trasy z zachowaniem podanych powyżej warunków jest możliwe.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego TAN.IN dane: długość trasy, liczbę hoteli oraz oferty hoteli;*
- *znajduje dwa plany podróży:*
  - **najtańszej**, *tzn. takiej, żeby suma opłat za hotele była najmniejsza, a jeśli takich rozwiązań jest wiele, wybiera jedno z najmniejszą liczbą noclegów,*
  - **najszybszej**, *tzn. takiej, żeby liczba noclegów była najmniejsza, a jeśli takich rozwiązań jest wiele, wybiera jedno z najmniejszą sumą opłat za noclegi;*
- *zapisuje wyniki, tj. dwa plany podróży — najtańszej i najszybszej, w pliku tekstowym TAN.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego TAN.IN są zapisane dwie liczby całkowite dodatnie, oddzielone pojedynczym odstępem: długość trasy d w kilometrach oraz liczba hoteli h, gdzie $d \leqslant 16000$, $h \leqslant 1000$.
W kolejnych h wierszach są zapisane oferty hoteli — każda oferta w osobnym wierszu. Są one uporządkowane według rosnącej odległości hoteli od początku trasy. Każda oferta jest zapisana w postaci dwóch liczb całkowitych dodatnich, oddzielonych pojedynczym odstępem, pierwsza liczba — to odległość hotelu od początku trasy w kilometrach, a druga — to cena jednego noclegu w tym hotelu nie większa niż 1000.

Wyjście

W pierwszym wierszu pliku tekstowego TAN.OUT należy zapisać plan podróży najtańszej — odległość kolejnych miejsc noclegu od początku trasy.
W drugim wierszu należy — w taki sam sposób — zapisać plan podróży najszybszej.
Liczby w wierszu powinny być pooddzielane pojedynczym odstępem.

Przykład

Dla pliku TAN.IN

```
2000 7
100 54
120 70
400 17
700 38
1000 25
1200 18
1440 40
```

poprawnym rozwiązaniem jest plik TAN.OUT mający dwa identyczne wiersze:

```
400 1200
400 1200
```

Twój program powinien szukać pliku TAN.IN w katalogu bieżącym i tworzyć plik TAN.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę TAN.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku TAN.EXE

ROZWIĄZANIE

Przy omawianiu zadania będą używane następujące oznaczenia:

- D — długość trasy,
- H — liczba hoteli na trasie,
- M — maksymalny zasięg dzienny,
- o_i — odległość i-tego hotelu od początku trasy,
- c_i — cena noclegu w i-tym hotelu,

Hotele będą numerowane od 1 do H, początek trasy będzie traktowany jak hotel numer 0, a koniec — jak hotel numer $H+1$. Oczywiście $c_0 = c_{H+1} = 0$, $o_0 = 0$, a $o_{H+1} = D$.

Najlepszą metodą rozwiązania obu problemów zadania jest programowanie dynamiczne. Dla każdego hotelu i liczony jest następnik n_i oraz wartość k_i — para (liczba noclegów, pieniądze) — minimalny koszt dojazdu od niego do końca trasy. Dla hotelu końcowego następnik jest nieokreślony, a koszt $k_{H+1} = (0,0)$. Dla każdego hotelu i od H do 0, wartość k_i liczy się jako minimalny koszt dojazdu od pierwszego noclegu do końca, powiększony o koszt spędzenia nocy w hotelu i, co się wyraża wzorem:

$$k_i = \min_{j:\ i<j\leq H+1\ \wedge\ o_j-o_i\leqslant M} k_j + (1, c_i)$$

przy czym minimum jest liczone względem współrzędnej odpowiadającej wersji zadania (liczba noclegów lub pieniądze). Następnik n_i to oczywiście ten hotel z rozpatrywanego zbioru, którego koszt był minimalny. Wartość $k_0 - (1,0)$ to wartość minimalnego kosztu przejazdu*, a ciąg $(n_0, n_{n_0}, \ldots)$ jest poszukiwanym rozwiązaniem. Wszystkie operacje na parach dokonujemy po współrzędnych.

W zależności od wersji szukanej trasy (najtańsza, najkrótsza) zmienia się tylko sposób obliczania funkcji minimum. Ponieważ kolejno liczone od końca wartości k_i są zapamiętywane, dla każdego hotelu wystarczy przejrzeć tylko te hotele, które są od niego oddalone nie więcej niż o 800 km, a jest ich nie więcej niż 800. Dzięki temu pesymistyczna złożoność czasowa tego algorytmu wynosi $O(HM)$.

Można by jeszcze pokusić się o zmniejszenie złożoności czasowej do $O(H \log M)$. Zbiór $\{j : i < j \leqslant H+1 \wedge o_j - o_i \leq M\}$, z którego wybiera się kolejne minima, modyfikuje się bardzo nieznacznie przy zmianie i, a więc gdyby go zaimplementować np. jako drzewo AVL, czasy wyboru minimum, usunięcia hoteli „wystających” poza 800 km, wstawienia nowego hotelu, byłyby rzędu $O(\log M)$. Ale ponieważ

* Trzeba odjąć nocleg w hotelu 0, który się nie liczy.

zgodnie z treścią zadania M jest stałe, a narzut na wykonywanie operacji na drzewie AVL byłby niemały, więc zmiana ta wcale nie musiałaby spowodować istotnego przyśpieszenia.

Minimalna złożoność pamięciowa tego problemu wynosi oczywiście $\Theta(H)$, gdyż dla każdego hotelu musimy zapamiętać przynajmniej jego odległość i cenę. Rozwiązanie opisane powyżej ma właśnie złożoność liniową. (Zobacz program TAN.PAS na dołączonej dyskietce.)

Rozwiązanie to można by zrealizować za pomocą procedur rekurencyjnych z zapamiętywaniem wyników poprzednich wywołań.

Do rozwiązania tego zadania można użyć wielu znanych algorytmów grafowych. Traktują one trasę z hotelami, jak graf i wyszukują w nim odpowiednie ścieżki. Wierzchołkami grafu są oczywiście wszystkie hotele oraz punkt początkowy i końcowy, a skierowanymi krawędziami połączone są tylko te hotele, których wzajemna odległość nie przekracza 800 km. Rozwiązaniem zadania będą dwie ścieżki łączące wierzchołek początkowy z końcowym: pierwsza — o najmniejszej długości, a druga — o najmniejszej sumie cen noclegów. Skuteczność programów zależy od użytego algorytmu szukania optymalnej ścieżki. Należy przy tym tak zaprojektować algorytm, by nie trzeba było liczyć i zapamiętywać wszystkich krawędzi, gdyż ich liczba może być bardzo duża!

Na podstawie algorytmu wyszukiwania najkrótszych ścieżek, opisanego np. w książce [10], można skonstruować rozwiązanie równie skuteczne, co programowanie dynamiczne. Tutaj również liczony jest koszt dojazdu od danego hotelu do końca trasy, ale w odwrotny sposób.

Podobnie, jak w przedstawionym rozwiązaniu, z każdym* hotelem i związane są dwie wartości: tymczasowy koszt minimalny dojazdu do końca k_i — para (liczba noclegów, pieniądze), początkowo $(+\infty, +\infty)$ — oraz następnik n_i — początkowo nieokreślony. Następnie końcowi trasy — hotelowi numer $H+1$ — nadane są wartości początkowe (0,0). Potem dla każdego hotelu i od $H+1$ do 1, bada się wszystkie jego poprzedniki j, czy dotychczasowa wartość kosztu minimalnego k_j nie jest większa od kosztu jaki byłby potrzebny na przejazd, gdyby pierwszy nocleg był spędzony w hotelu i. Po dojechaniu do hotelu 1 wartość $k_0 - (1, 0)$, podobnie jak w rozwiązaniu wzorcowym, będzie minimalnym kosztem dojazdu od początku trasy do końca, a ciąg $(n_0, n_{n_0}, \ldots)$ — ciągiem numerów hoteli na optymalnej trasie.

Oczywiście wyłącznie od wyboru metody porównywania kosztów zależy, czy znalezione rozwiązanie będzie najtańszym czy najkrótszym. Koszt czasowy tego rozwiązania będzie wynosił $O(HM)$ i tym razem nie da się go w łatwy sposób poprawić. Koszt pamięciowy oczywiście jest również $O(H)$.

Metoda płonącego lasu

Ta metoda opiera się na odmiennej filozofii od rozwiązań poprzednich. Rozważamy tu największy zbiór hoteli, z których można dojechać do końca trasy za określoną pulę noclegów i pieniędzy. Następnie stopniowo powiększamy tę pulę, aż wierzchołek końcowy wpadnie do rozważanego zbioru.

Podobnie jak w dwóch poprzednich rozwiązaniach, dla każdego hotelu i określamy tymczasowy minimalny koszt (liczba noclegów, pieniądze) dojazdu od niego do końca trasy k_i, na początku równy $(+\infty, +\infty)$ dla każdego wierzchołka, oprócz wierzchołka końcowego, dla którego $k_{H+1} = (0, 0)$. Dla każdego wierzchołka i, n_i oznacza jego następnika (na początku nieokreślony). Określamy zbiór Q hoteli o ustalonym minimalnym koszcie dojazdu do końca, który na początku jest pusty.

Do zbioru Q dorzucamy ten wierzchołek $i \notin Q$, którego koszt k_i jest najmniejszy (jako pierwszy oczywiście zostanie dorzucony $H+1$). Dla wszystkich jego poprzedników j (hoteli odległych o nie więcej niż 800 km) weryfikuje się wartości k_j i n_j, sprawdzając, czy koszt dojazdu z wierzchołka j do końca z pierwszym noclegiem w dorzuconym właśnie wierzchołku i nie byłby niższy od dotychczasowej wartości. Powyższy krok powtarza się tak długo, aż wierzchołek początkowy 0 zostanie dorzucony do zbioru Q. Wówczas, podobnie jak dotąd, ciąg $(n_0, n_{n_0}, \ldots)$ to lista hoteli na ścieżce od 0 do $H+1$ o najmniejszym koszcie. Szczegółowy opis oraz dowód poprawności przedstawionego właśnie **algorytmu Dijkstry** można znaleźć w książce [10].

Powyższy algorytm stosuje się oczywiście do obu wersji, w zależności od sposobu porównywania par (noclegi, pieniądze). Jego pesymistyczna złożoność czasowa wynosi $O(H^2M)$, ale dałoby się ją poprawić.

* Fakt, że hotele są tu w kolejności jest istotny. W ogólności pierwszym krokiem algorytmu jest sortowanie topologiczne wierzchołków grafu.

Użycie bardziej skomplikowanych struktur danych do przechowywania zbioru Q jako kolejki priorytetowej zmniejszyłoby złożoność do $O(MH \log H)$. Złożoność pamięciowa jest rzędu H, czyli optymalna.

Przeszukiwanie grafu wszerz i w głąb

Jeśli graf nie jest fizycznie konstruowany, to algorytmy te odpowiadają rozwiązaniom, w którym w ogóle nie występuje pojęcie grafu. Przeszukiwanie w głąb odpowiada przeglądaniu wszystkich lub niektórych ścieżek, a przeszukiwanie wszerz — jakiejś wersji programowania dynamicznego

Następniki czy poprzedniki?

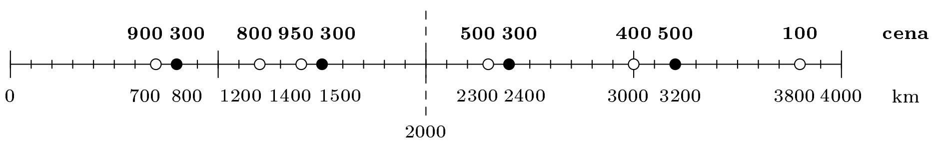
Podane dotychczas rozwiązania mogłyby zamiast następników i kosztów dojazdu od początku trasy liczyć dla każdego hotelu jego poprzednik oraz koszt dojazdu do końca trasy. Jednak musielibyśmy wtedy odwrócić obliczony ciąg poprzedników, czyli potrzebna byłaby dodatkowa pętla.

Badanie wszystkich możliwych rozkładów

Najbardziej oczywiste rozwiązanie polegało oczywiście na badaniu wszystkich możliwych rozkładów. Niestety liczba tych rozkładów może być tak wielka, że rozwiązanie takie jest nieakceptowalne, niezależnie czy implementacja będzie rekurencyjna czy iteracyjna. Pomimo ograniczenia liczby badanych rozkładów niezadowalające są również ulepszone wersje, np. branie pod uwagę rozkładów o niezbyt dużej liczbie noclegów (nie ma sensu nocować więcej niż $2 * D/M$ razy), lub wyłącznie takich rozkładów noclegów $(i_1, i_2, \ldots, i_n)$, w których dla dowolnego j mamy $o_{i_{j+1}} - o_{i_{j-1}} > 800$ (nie zatrzymujemy się, jeżeli tego samego dnia można dojechać do następnego planowanego miejsca postoju).

Podział na podzadania

Dobrą metodą rozwiązania wielu podobnych na pierwszy rzut oka zadań jest podział całości na części i rozwiązywanie każdej z nich oddzielnie, a następnie połączenie rozwiązań poszczególnych części. Jednak w tym zadaniu, metoda ta nie przynosiła oczekiwanych rezultatów. Na rysunku poniżej optymalne noclegi dla każdej z połówek są oznaczone na czarno, a optymalny rozkład dla całej trasy to noclegi czarne z części lewej i białe z prawej:



TESTY

Rozwiązania zawodników sprawdzano za pomocą trzech rodzajów testów: testów o niewielkiej liczbie hoteli (TAN0.IN–TAN5.IN), testów do badania przypadków granicznych (TAN6.IN–TAN8.IN) oraz testów trudnych z długimi trasami i dużą liczbą hoteli (TAN9.IN–TAN11.IN).

- TAN0.IN — przykład z treści zadania
- TAN1.IN — test pokazujący, że nie zawsze z rozwiązań częściowych da się w prosty sposób złożyć rozwiązanie całości.
- TAN2.IN — test mający na celu odrzucenie programów przeszukujących wszystkie ścieżki (ponad 10^{20}, gdy ilość ścieżek minimalnych dla tego testu wynosi 10^7);

- TAN3.IN — test na poprawność implementacji algorytmu dla dużych liczb;
- TAN4.IN — test poprawnościowy;
- TAN5.IN — test poprawnościowy; optymalny rozkład noclegów obejmuje same najdroższe hotele;
- TAN6.IN — test mający na celu odrzucenie rozwiązań budujących w pamięci graf hoteli (ponad 400 tys. połączeń).
- TAN7.IN — test poprawnościowy; na trasie o długości 700 km znajduje się 500 hoteli, w żadnym nie trzeba nocować;
- TAN8.IN — test poprawnościowy; jedyny hotel w środku trasy o długości 1600 km;
- TAN9.IN–TAN11.IN — testy wydajnościowe.

Bramki XOR

Każda **bramka XOR** *ma dwa wejścia i jedno wyjście, a jej działanie opisuje następująca tabelka:*

wejście 1	*wejście 2*	*wyjście*
0	*0*	*0*
0	*1*	*1*
1	*0*	*1*
1	*1*	*0*

Siecią XOR *nazywamy układ bramek XOR, mający n wejść i jedno wyjście, spełniający następujące warunki:*

(1) Każde wejście sieci XOR jest połączone z przynajmniej jednym wejściem bramki.

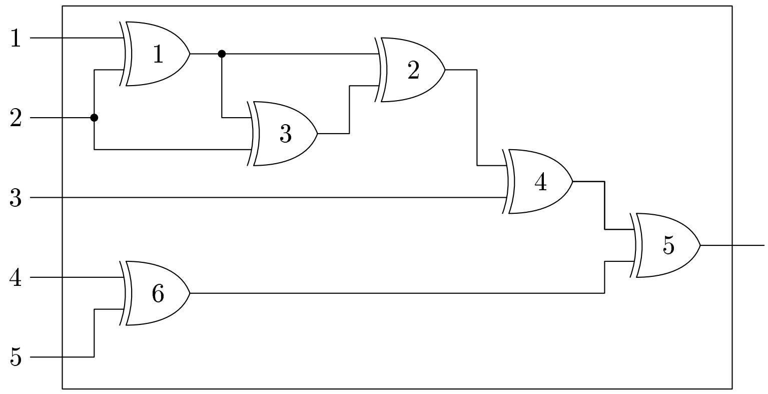
(2) Każde wejście każdej bramki jest połączone z jednym wejściem sieci albo z jednym wyjściem innej bramki.

(3) Wyjście dokładnie jednej bramki jest połączone z wyjściem sieci.

(4) Każde wyjście bramki w sieci jest połączone z przynajmniej jednym wejściem innej bramki albo z jedynym wyjściem sieci.

(5) Istnieje taka numeracja bramek, że do każdego wejścia dowolnej bramki jest podłączone wejście sieci albo wyjście bramki o mniejszym numerze.

Przykład



Przedstawiony na rysunku układ 6 bramek mający 5 wejść i 1 wyjście spełnia warunki 1–5, więc jest siecią XOR.

UWAGA: *Bramki na rysunku zostały ponumerowane dowolnie, ale istnieje numeracja spełniająca warunek określony w punkcie 5.*

Wszystkie wejścia sieci są ponumerowane od 1 do n. Stan wejść sieci XOR opisuje **słowo wejściowe** *utworzone z n cyfr dwójkowych 0 i 1 — przyjmujemy, że i-ta od lewej cyfra danego słowa wejściowego, to stan i-tego wejścia sieci. Dla dowolnego stanu wejść sieć daje na wyjściu 0 albo 1. Każde słowo wejściowe jest dwójkowym zapisem jakiejś liczby naturalnej, więc słowa te można uporządkować zgodnie z ich wartościami liczbowymi. Sieci XOR będziemy testowali podając na wejściu kolejne słowa z ustalonego zakresu i zliczając liczbę otrzymanych w wyniku jedynek.*

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego XOR.IN opis sieci XOR: liczbę wejść n, liczbę bramek m, numer bramki połączonej z wyjściem sieci oraz opisy połączeń, a następnie dwa n-bitowe słowa wejściowe — dolne i górne ograniczenie zakresu, w jakim będziemy testowali sieć,*
- *oblicza liczbę jedynek otrzymanych na wyjściu sieci dla słów wejściowych z danego zakresu,*
- *zapisuje wynik w pliku tekstowym XOR.OUT.*

Zakładamy, że $3 \leqslant n \leqslant 100$, $3 \leqslant m \leqslant 3000$ oraz, że bramki danej sieci są ponumerowane w dowolnym porządku liczbami od 1 do m.

Wejście

W pierwszym wierszu pliku tekstowego XOR.IN są zapisane trzy liczby całkowite dodatnie pooddzielane pojedynczym odstępem. Jest to liczba wejść n danej sieci XOR, liczba bramek m oraz numer bramki połączonej z wyjściem sieci. W kolejnych m wierszach znajdują się opisy połączeń bramek sieci. W i-tym z tych wierszy, dla i od 1 do m, znajduje się opis połączeń dwóch wejść bramki o numerze i, który ma postać dwóch liczb całkowitych nie mniejszych niż $-n$ i nie większych niż m, oddzielonych pojedynczym odstępem. Jeśli odpowiednie wejście do bramki jest połączone z wejściem do sieci o numerze k, to opisem tego połączenia jest liczba ujemna $-k$, a jeśli wejście do bramki jest połączone z wyjściem innej bramki o numerze j, to opisem tego połączenia jest liczba dodatnia j.
W kolejnych 2 wierszach pliku tekstowego XOR.IN są zapisane dwa n-bitowe słowa $\underline{a}$ oraz $\underline{b}$. Jest to dolne i górne ograniczenie zakresu testowania sieci. Zakładamy, że w danym zakresie mieści się nie więcej niż sto tysięcy słów.

Wyjście

W pliku tekstowym XOR.OUT należy zapisać jedną liczbę całkowitą nieujemną — liczbę jedynek, jakie powinniśmy otrzymać na wyjściu poprawnie działającej danej sieci XOR dla słów wejściowych $\underline{s}$ z danego zakresu $\underline{a} \leqslant \underline{s} \leqslant \underline{b}$, gdzie nierówność $\leqslant$ należy rozumieć jako relację porządku zgodnego z wartościami liczbowymi słów dwójkowych.

Przykład

Dla pliku XOR.IN, zawierającego opis przedstawionej powyżej sieci XOR:

```
5 6 5
-1 -2
1 3
1 -2
2 -3
4 6
-4 -5
00111
01110
```

poprawnym rozwiązaniem jest następujący plik XOR.OUT:

```
5
```

Twój program powinien szukać pliku XOR.IN w katalogu bieżącym i tworzyć plik XOR.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę XOR.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku XOR.EXE

ROZWIĄZANIE

Kluczem do efektywnego rozwiązania jest znalezienie zależności pomiędzy wartościami na wejściu sieci, a wynikiem na jej wyjściu. W tym celu przyjrzyjmy się bliżej działaniu bramki XOR. Niech $\oplus$ oznacza dodawanie liczb całkowitych modulo 2, tzn. takie, że jego wynikiem jest reszta, jaką daje suma ze zwykłego dodawania przy dzieleniu przez 2. Tak określone działanie jest przemienne i łączne, więc możemy go używać analogicznie do zwykłego dodawania. Stan na wyjściu bramki (lub inaczej: liczbę dwójkową obliczaną przez bramkę) można teraz wyrazić jako $a \oplus b$, gdzie a i b oznaczają wartości na jej wejściach.

W ogólności, jeżeli przez $x_1, \ldots, x_n$ oznaczymy stany wejść sieci, to stan na wyjściu każdej z bramek zadanej sieci można wyrazić jako wartość funkcji postaci

$$f(x_1, \ldots, x_n) = a_1x_1 \oplus a_2x_2 \oplus \cdots \oplus a_nx_n \tag{1}$$

gdzie zapis a_1x_1 oznacza zwykłe mnożenie, a $a_1, \ldots, a_n \in \{0, 1\}$ są współczynnikami zależnymi od położenia bramki w sieci. W szczególności stan na wyjściu bramki połączonej z wyjściem sieci jest także opisany powyższą zależnością, a jej funkcję nazywamy **wielomianem sieciowym**. Dla przykładowej sieci z treści zadania wielomiany dla odpowiednich bramek mają postać (f_i oznacza wielomian dla i-tej bramki):

$$\begin{aligned}
f_1(x_1, \ldots, x_5) &= x_1 \oplus x_2 \\
f_2(x_1, \ldots, x_5) &= x_2 \\
f_3(x_1, \ldots, x_5) &= x_1 \\
f_4(x_1, \ldots, x_5) &= x_2 \oplus x_3 \\
f_5(x_1, \ldots, x_5) &= x_2 \oplus x_3 \oplus x_4 \oplus x_5 \\
f_6(x_1, \ldots, x_5) &= x_4 \oplus x_5
\end{aligned}$$

Jak znaleźć wielomian sieciowy? Po przyjrzeniu się postaci wielomianu (1) łatwo zauważyć, że dla $x_i = 1$ oraz $x_j = 0$ dla wszystkich $j \neq i$ wartość na wyjściu sieci jest tożsama ze współczynnikiem a_i jej wielomianu. Aby wyznaczyć cały wielomian, wystarczy obliczyć, jakie wyniki daje sieć na wyjściu dla n słów wejściowych o postaci podanej wyżej dla $i = 1, \ldots, n$. Powyższe obliczenie można prosto zaimplementować w postaci procedury rekurencyjnej wyznaczającej wartość na wyjściu bramki na podstawie wartości na wyjściach bramek podłączonych do jej wejść lub wartości na wejściach sieci. Przyjmijmy następujące globalne deklaracje:

```
const
  MaxLiczbaBramek = 3000;
  MaxLiczbaWejsc = 100;
type
  { Stan binarny }
  TStan = 0..1;
  { Element oznaczający bramkę lub wejście sieci }
  TElement = record
    { Numery bramek podłączonych do wejść; nieistotne w przypadku wejść sieci }
    Wejscie1, Wejscie2 : Integer;
    { Stan na wyjściu bramki lub stan wejścia sieci }
    Wyjscie : TStan;
    { Oznacza, czy wartość Wyjscie jest wyznaczona }
    WyjscieObliczone : Boolean
  end;
  { Sieć złożona z bramek i wejść }
  TSiec = array [-MaxLiczbaWejsc..MaxLiczbaBramek] of TElement;
  { Współczynniki wielomianu lub stan wejść }
  TWielomian = array [1..MaxLiczbaWejsc] of TStan;
var
  S : TSiec; { Cała sieć }
  N : Integer; { Liczba wejść }
  M : Integer; { Liczba bramek }
  WyjscieSieci : Integer; { Numer bramki podłączonej do wyjścia sieci }
```

Elementy $S[-N]..S[-1]$ opisują odpowiednie wejścia sieci, natomiast elementy $S[1]..S[M]$ przechowują dane o bramkach sieci. Poniższa procedura oblicza wartość na wyjściu bramki o numerze *nr* dla danego stanu wiejść $S[-N]$..S[−1]:

```
procedure ObliczBramke (nr : Integer);
```

```
var
  We1, We2 : Integer;
begin
  We1 := S[nr].Wejscie1; We2 := S[nr].Wejscie2;
  { Oblicz wartości wyjściowe bramek podłączonych do wejść,
    jeżeli nie są jeszcze wyznaczone }
  if not S[We1].WyjscieObliczone then ObliczBramke(We1);
  if not S[We2].WyjscieObliczone then ObliczBramke(We2);
  { Oblicz wartość na wyjściu }
  S[nr].Wyjscie := (S[We1].Wyjscie + S[We2].Wyjscie) mod 2;
  { Zaznacz, że wartość na wyjściu wyznaczona }
  S[nr].WyjscieObliczone := true
end;
```

Zauważmy, że dzięki pamiętaniu raz obliczonej wartości na wyjściu bramki, liczba kroków obliczeń wykonywanych przy wywołaniu *ObliczBramke*(*WyjscieSieci*) będzie proporcjonalna do liczby bramek m. Zauważmy także, że dzięki warunkowi 5. dla sieci XOR z treści zadania, takie wywołanie zakończy się sukcesem (nie będzie zapętlenia). Możemy teraz, wykorzystując procedurę `ObliczBramke`, łatwo wyznaczyć wielomian sieciowy:

```
var
  WielomianSieci : TWielomian; { Współczynniki wielomianu sieci }
  for i := 1 to N do begin
  { S[−N].Wyjscie..S[−1].Wyjscie := 0, S[−i].Wyjscie := 1 }
  { S[−N].WyjscieObliczone..S[−1].WyjscieObliczone := true }
  { S[1].WyjscieObliczone..S[M].WyjscieObliczone := false }
  { Oblicz wartość na wyjściu sieci... }
  ObliczBramke(WyjscieSieci);
  { ...która jest i-tym współczynnikiem }
  WielomianSieci[i] := S[WyjscieSieci].Wyjscie
end
```

Dobrze zaimplementowany powyższy fragment programu wykonuje liczbę operacji rzędu $m \cdot n$. Obliczenie liczby jedynek dla zadanego przedziału danych wejściowych wydaje się teraz proste:

```
var
  Dane, Koniec : TWielomian; { Słowa z pliku wejściowego }
  LiczbaJedynek : Longint; { Zlicza jedynki }
...
  LiczbaJedynek := 0;
  { Wczytaj wartości graniczne z pliku wejściowego na zmiene Dane i Koniec }
  LiczbaJedynek := LiczbaJedynek + WartoscWielomianu(WielomianSieci, Dane);
  while not RowneDane(Dane, Koniec) do begin
    KolejnyZestaw(Dane);
  LiczbaJedynek := LiczbaJedynek + WartoscWielomianu(Dane)
end;
```

Po uzupełnieniu implementacji procedur: *WartoscWielomianu* — obliczającej wartość wielomianu dla zadanego słowa, *RowneDane* — stwierdzającej, czy dwa słowa są równe, *KolejnyZestaw* — generującej kolejny zestaw wartości wejściowych zgodnie z porządkiem liczb dwójkowych oraz po zaimplementowaniu wczytania danych o sieci, otrzymujemy gotowy program, który można znaleźć na dołączonej dyskietce.

Złożoność czasowa przedstawionego rozwiązania jest rzędu $mn + ln$, gdzie l oznacza liczbę zestawów wejściowych, natomiast wymagania pamięciowe są rzędu $m + n$.

Możemy sobie wyobrazić, że procedura rekurencyjna podobna do *ObliczBramke* będzie wyznaczać nie wartość na wyjściu bramki lecz cały jej wielomian, na podstawie wielomianów bramek podłączonych do jej wejść lub wielomianów odpowiadającym wejściom sieci. Oczywiście wielomianem odpowiadającym i-temu wejściu sieci jest x_i. Złożoność czasowa rozwiązania opartego na tym pomyśle jest tego rzędu, co złożoność rozwiązania poprzedniego, lecz potrzebna pamięć jest rzędu mn na przechowywanie wielomianów.

Jeżeli wyeliminujemy pamiętanie wyników pośrednich (wyjście bramki lub wielomian), przy zastosowaniu procedury podobnej do *ObliczBramke* otrzymamy nieco prostszy algorytm, za to bardzo nieefektywny (dane dla jednej bramki wyliczane byłyby wielokrotnie).

Dane dla bramki (wyjście bramki lub wielomian) można wyznaczyć w inny sposób niż przez stosowanie rekurencji. W tym celu dla bramek zadanej sieci zdefiniujmy relację $<$ (mniejsza) w ten sposób, że bramka B_1 jest mniejsza od bramki B_2 ($B_1 < B_2$), jeżeli wyjście bramki B_1 podłączone jest do jednego z wejść bramki B_2. Zgodnie z warunkiem 5. dla sieci XOR z treści zadania, jej bramki można ustawić w ciąg tak, aby na prawo od danej bramki w ciągu nie występowały bramki od niej mniejsze. Łatwo zauważyć, że na początku ciągu będzie bramka, której oba wejścia są podłączone do wejść sieci, a na końcu ta, której wyjście jest podłączone do wyjścia sieci. Znalezienie takiego uporządkowania nazywa się **sortowaniem topologicznym**. Z odpowiednim algorytmem i jego omówieniem Czytelnik może się zapoznać np. w książce [21]. Gdy już znajdziemy takie uporządkowanie, wystarczy obliczyć kolejno wartości na wyjściach dla każdej bramki w ciągu. Złożoność czasowa rozwiązania opartego na tym pomyśle jest taka sama, jak rozwiązania pierwszego (!), a złożoność pamięciowa zależy od tego, czy obliczamy wielomian sieci przez symulację, czy przez wyznaczanie wielomianów dla każdej bramki.

Możemy wreszcie zrezygnować z wyznaczania wielomianu sieci i obliczać wartość na jej wyjściu, dla każdych danych wejściowych, stosując rekurencję z pamiętaniem wyników pośrednich lub bez, bądź też sortowanie topologiczne. Jednak złożoność czasowa takich rozwiązań będzie przynajmniej rzędu lm, a ponieważ m może być dużo większe niż n, przy dużych l rozwiązania takie będą znacznie wolniejsze niż rozwiązanie zaproponowane jako pierwsze.

TESTY

Do sprawdzania rozwiązań zawodników użyto 10 testów XOR0.IN–XOR9.IN. Testy można podzielić na dwie grupy: testy sprawdzające szczególne przypadki oraz testy sprawdzające szybkość działania zastosowanego algorytmu. Oto krótki ich opis:

- XOR0.IN — test z treści zadania;
- XOR1.IN — test sprawdzający zachowanie programu, gdy występują w sieci bramki o obu wejściach podłączonych do tego samego punktu;
- XOR2.IN — dla tego testu odpowiedzią jest 0 jedynek;
- XOR3.IN — dla tego testu odpowiedź przekracza zakres typu **Word**.
- XOR4.IN — prosty test sprawdzający poprawność użytej metody;

Testy XOR5.IN–XOR9.IN to testy złożnościowe o wzrastającym stopniu trudności.