Zawody III stopnia

Zawody III stopnia — opracowania zadań

Marcin Sawicki
Treść zadania, Opracowanie

Andrzej Gąsienica–Samek
Program

Wędrowni treserzy pcheł

W Bajtocji można spotkać wędrownych treserów pcheł. Tresowane pchły uczone są tańca, polegającego na wykonywaniu precyzyjnych skoków w rytm muzyki. Dokładnie wygląda to tak: treser układa na stole w rządku ponumerowane żetony, przy czym żetony nie muszą być ułożone po kolei. Na każdym żetonie, oprócz jego numeru, jest również napisany numer żetonu, na który powinna z niego skoczyć pchła. Następnie treser ustawia po jednej pchle na każdym z żetonów i włącza muzykę. Na początku każdego taktu, każda z pcheł wykonuje skok wprost na żeton, którego numer jest napisany na żetonie, na którym w danej chwili stoi. W trakcie tańca może się zdarzyć, że kilka pcheł znajdzie się na tym samym żetonie i razem wykonują dalsze skoki.

Załóżmy, że mamy n tresowanych pcheł i n żetonów. Jeśli podamy, jakie liczby znajdują się kolejno na żetonach numer $1, 2, \ldots, n$, to jednoznacznie opiszemy układ choreograficzny, jaki zaprezentują pchły. Jednak może się okazać, że dwa różne zestawy żetonów dają ten sam układ, jeśli tylko odpowiednio je ułożymy.

Przykład

Powiedzmy, że mamy trzy żetony. Jeśli z żetonu nr 1 należy skoczyć na żeton nr 2, z żetonu nr 2 na żeton nr 3, a z żetonu nr 3 na żeton nr 1 (w skrócie: $1 \to 2,\ 2 \to 3,\ 3 \to 1$), to pchły będą tańczyć „w kółko" i żadne dwie nigdy się nie spotkają na tym samym żetonie. Jest to inny układ tańca, niż np. $1 \to 2,\ 2 \to 3,\ 3 \to 3$, gdzie już po dwóch taktach wszystkie trzy pchły spotkają się na żetonie nr 3 i dalej będą razem skakać w miejscu.

Natomiast układy $1 \to 2,\ 2 \to 3,\ 3 \to 2,\ 4 \to 4$ oraz $1 \to 1,\ 2 \to 3,\ 3 \to 2,\ 4 \to 3$ są takie same — wystarczy ułożyć żetony na stole w rzędzie, w pierwszym przypadku w kolejności od lewej do prawej, a w drugim od prawej do lewej, a pchły odtańczą ten sam taniec.

Zadanie

Gawiedź bardzo się niecierpliwi, gdy pchły tańczą według tego samego układu więcej niż raz. Dlatego potrzebny jest program, który:

- *wczyta z pliku tekstowego* `pch.in` *liczbę przypadków testowych,*
- *dla każdego z przypadków wczyta z pliku* `pch.in` *opis dwóch zestawów żetonów i rozstrzygnie, czy żetony z tych zestawów można ułożyć na stole tak, by pchły wykonały taki sam taniec,*
- *wypisze odpowiedzi do pliku tekstowego* `pch.out`.

Wejście

W pierwszym wierszu pliku tekstowego `pch.in` *znajduje się jedna liczba całkowita d równa liczbie przypadków testowych, $1 \le d \le 100$.*

Kolejne $3d$ wierszy pliku `pch.in` *opisują kolejne przypadki testowe — każdy przypadek zajmuje trzy kolejne wiersze pliku. Pierwszy z nich zawiera jedną liczbę całkowitą $1 \le n \le 2\,000$, równą liczbie żetonów. Każdy z dwóch następnych wierszy zawiera opis zestawu n żetonów w postaci ciągu n liczb całkowitych z przedziału $1 \ldots n$, pooddzielanych pojedynczymi odstępami; i-ty wyraz ciągu oznacza numer żetonu, na który mają skakać pchły z żetonu nr i.*

Wyjście

Dla każdego z przypadków testowych z pliku `pch.in` *należy wypisać do pliku tekstowego* `pch.out` *dokładnie jeden wiersz, zawierający dokładnie jedną literę:*

- *T — jeśli oba zestawy żetonów można ułożyć tak, aby pchły wykonały taki sam taniec,*
- *N — w przeciwnym wypadku.*

Przykład

Dla pliku wejściowego `pch.in`*:*

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

poprawną odpowiedzią jest plik wyjściowy `pch.out`*:*

```
N
T
```

Rozwiązanie

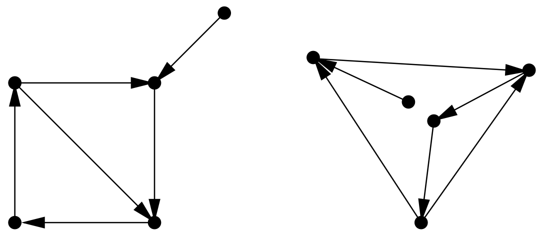
Wypada nam zacząć od przypomnienia, co to jest graf.

Graf są to kropki, z których niektóre połączone są strzałkami. Albo inaczej: graf G jest to para uporządkowana $G = \langle V, E\rangle$, gdzie V jest dowolnym zbiorem, który nazwiemy *zbiorem wierzchołków* grafu, zaś $E \subseteq V \times V$ jest *zbiorem krawędzi*: jeżeli $v, v' \in V$ są dwoma wierzchołkami grafu, to są one połączone strzałką od v do v' wtedy i tylko wtedy, gdy para $(v, v') \in E$.

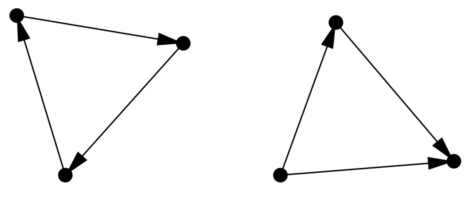
Zwróćmy uwagę, że w przyjętej przez nas definicji krawędź od v do v' to nie to samo, co krawędź od v' do v. Ponadto, dopuszczamy *pętle*, czyli krawędzie prowadzące od wierzchołka do niego samego.

Grafy są przydatne do opisu wielu praktycznych zagadnień, np. sieci komunikacyjnych, obwodów elektrycznych, zależności pomiędzy etapami przedsięwzięcia, a także, jak się zaraz okaże, pchlej choreografii.

Kiedy rysujemy graf, zazwyczaj nie ma dla nas znaczenia, jak na naszym rysunku rozmieszczone są wierzchołki, ważne jest tylko, które z którymi są połączone. Poza tym często wszystkie wierzchołki grafu zaznaczamy tak samo (np. kropką). Stąd pojawia się problem *izomorfizmu grafów*, polegający na rozstrzygnięciu, czy dane dwa grafy $G_1 = \langle V_1, E_1\rangle$, $G_2 = \langle V_2, E_2\rangle$ da się narysować tak samo? Innymi słowy, chodzi o sparowanie wierzchołków grafu G_1 z wierzchołkami grafu G_2 tak, by krawędzie w grafie G_1 dokładnie odpowiadały krawędziom pomiędzy wierzchołkami do pary w grafie G_2. Jeszcze inaczej, chodzi o znalezienie funkcji wzajemnie jednoznacznej $f : V_1 \longrightarrow V_2$ takiej, by $(v, v') \in E_1$ zachodziło wtedy i tylko wtedy, gdy $(f(v), f(v')) \in E_2$.



Rysunek 1: Para grafów izomorficznych



Rysunek 2: Para grafów nieizomorficznych

Wróćmy teraz do naszego zadania. Mamy opis pewnego tańca pcheł, czyli n żetonów ponumerowanych liczbami $1, 2, \ldots, n$, a na każdym z nich dodatkową liczbę mówiącą, na który żeton pchła ma skoczyć. Rozważmy graf, którego zbiorem wierzchołków będzie właśnie $V = \{1, 2, \ldots, n\}$, czyli każdemu żetonowi odpowiada dokładnie jeden wierzchołek grafu. Krawędzie poprowadzimy oczywiście od każdego żetonu do tego, którego numer jest na nim zapisany. Krawędzie będą zatem odpowiadać pchlim skokom.

Teraz naprawdę nietrudno zauważyć, że nasze zadanie to po prostu pytanie o izomorfizm tak utworzonych grafów! Zaglądamy zatem do indeksu dowolnego podręcznika algorytmiki, znajdujemy hasło „grafów izomorfizm” i ... co za rozczarowanie! Okazuje się, że nie jest znany żaden wielomianowy algorytm na rozstrzyganie o izomorfizmie grafów. Nikt też nie zdołał udowodnić, że taki algorytm nie może istnieć, co więcej nie wiadomo nawet, czy jest to tzw. problem NP–zupełny.[1]

Czyżby było aż tak źle? Pomyślmy... niektórzy spośród czytelników słyszeli może o problemie izomorfizmu drzew, dla którego istnieje rozwiązanie wielomianowe. Drzewa to grafy bez cykli. Być może grafy, które opisują pchle harce, też mają jakąś szczególną postać, która pozwala łatwo rozwiązać nasz problem? Okazuje się, że tak.

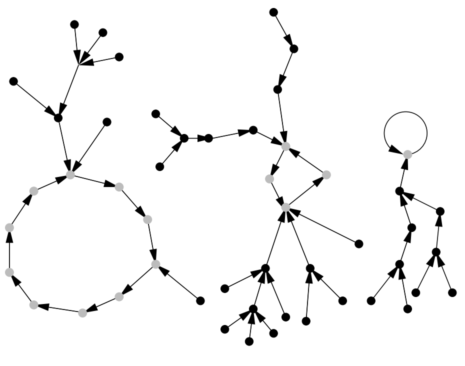
Przede wszystkim, w naszych grafach z każdego wierzchołka wychodzi *dokładnie* jedna krawędź. Oczywiście jest całe mnóstwo grafów, które nie mają tej właściwości, np. trzy spośród czterech narysowanych powyżej.

Zdefiniujmy *cykl* (skierowany) jako ciąg $(v_1, v_2, \ldots, v_n)$ parami różnych wierzchołków grafu $G = \langle V, E\rangle$ taki, że dla każdego $i = 1, 2, \ldots, n-1$ mamy w grafie krawędź $(v_i, v_{i+1}) \in E$ oraz dodatkowo $(v_n, v_1) \in E$. Zauważmy, że jeżeli ciąg $(v_1, v_2, \ldots, v_n)$ jest cyklem, to dla każdego $i = 2, 3, \ldots, n$ cyklem jest także ciąg $(v_i, v_{i+1}, \ldots, v_n, v_1, v_2, \ldots, v_{i-1})$, otrzymany przez cykliczne przesunięcie ciągu wyjściowego w lewo o $i-1$ pozycji. Umówmy się, że są to równoważne reprezentacje jednego i tego samego cyklu.

A co możemy powiedzieć o cyklach w naszych pchlich grafach? Wykażemy, że są one rozłączne, to znaczy że każdy wierzchołek należy do co najwyżej jednego cyklu. Dlaczego? Przypuśćmy, że wierzchołek $v = v_i = v'_j$ należy do dwóch cykli: $v_1, v_2, \ldots, v_n$ oraz $v'_1, v'_2, \ldots, v'_m$. Umówiliśmy się, że ciągi reprezentujące cykle możemy przesuwać cyklicznie, zatem nic nie stracimy zakładając, że $v = v_1 = v'_1$ (czyli że $i = j = 1$). Zamieniając w razie potrzeby cykle rolami możemy również przyjąć, że $n \le m$. Wiemy, że z wierzchołka v (podobnie jak z każdego innego) wychodzi dokładnie jedna krawędź, wiemy też, że mamy w naszym grafie krawędź (v_1, v_2) oraz (v'_1, v'_2). Skoro zatem $v = v_1 = v'_1$, to także $v_2 = v'_2$. Rozumując tak dalej wnioskujemy, że $v_3 = v'_3, \ldots, v_n = v'_n$. Jeżeli byłoby $m > n$, to mielibyśmy następnie $v'_{n+1} = v_1 = v'_1$, lecz jest to niemożliwe, bo umówiliśmy się, że żaden wierzchołek w cyklu się nie powtarza. Czyli $n = m$ i nasze cykle okazały się być jednym i tym samym cyklem (a nawet tą samą jego reprezentacją)!

Każdy wierzchołek w naszym grafie albo należy do jakiegoś jednego cyklu, albo nie należy do żadnego. Jak wygląda ten drugi przypadek? Niech wierzchołek v_1 nie należy do żadnego cyklu. Prowadzi z niego dokładnie jedna krawędź, umówmy się, że do wierzchołka v_2, z którego z kolei prowadzi krawędź do v_3 itd. Otrzymujemy ciąg wierzchołków $v_1, v_2, v_3, v_4, \ldots$. Wiemy, że v_1 występuje tylko na początku ciągu (skoro nie leży na żadnym cyklu). Jednak w naszym grafie jest tylko skończenie wiele wierzchołków, więc od któregoś miejsca muszą się one zacząć powtarzać. Niech n będzie najmniejszą taką liczbą, że istnieje $i < n$ takie, że $v_i = v_n$. Oczywiście $i > 1$. Chwila uwagi wystarcza by stwierdzić, że wierzchołki $(v_i, v_{i+1}, v_{i+2}, \ldots, v_{n-1})$ tworzą cykl. Być może $n = i + 1$, mamy wówczas cykl jednoelementowy, czyli pętlę przy wierzchołku v_i.

Podsumujmy: startując od dowolnego wierzchołka i podążając po strzałkach, albo jesteśmy od razu na cyklu, albo po jakimś czasie wpadamy na cykl. Sytuacja przypomina morza i rzeki: jeżeli jesteśmy w wodzie, to albo jesteśmy w morzu, i wtedy wiadomo w jakim, albo jesteśmy w rzece i w końcu spłyniemy do morza. Do tego wiadomo, w zlewisku jakiego morza leży dana rzeka. Co więcej, analogia obejmuje i to, że rzeka ma dopływy. Istotnie, ścieżki prowadzące od różnych wierzchołków do tego samego cyklu mogą się łączyć jeszcze przed osiągnięciem cyklu, tworząc podczepione do cykli drzewa. Przykładowo, wygląda to tak, jak na rysunku 3. Wierzchołki, które leżą na cyklach, zaznaczono na szaro.



Rysunek 3: Przykład pchlego grafu

[1]Problemy NP–zupełne to m.in. problem cyklu Hamiltona, problem komiwojażera, problem 3-kolorowania grafu, sumy podzbioru, maksymalnej kliki, spełnialności formuł boolowskich i kilkaset innych. Wiadomo o tych problemach tyle, że albo wszystkie mają rozwiązania wielomianowe, albo żaden z nich takiego rozwiązania nie ma. Ponieważ przez wiele lat nie wymyślono dla żadnego z nich algorytmu wielomianowego (za to wymyślono wiele naprawdę pomysłowych algorytmów dla innych problemów), więc większość informatyków przypuszcza, że takie algorytmy nie istnieją. Wciąż jednak nikt nie potrafi tego udowodnić. Więcej o problemach NP i NP–zupełnych można poczytać w znakomitej książce [**?**].

Wcześniej wspominaliśmy o problemie izomorfizmu drzew, dla którego znamy rozwiązanie w czasie wielomianowym. Rozwiązanie to polega na wyznaczeniu dla drzewa D jego sygnatury $\sigma(D)$, to znaczy takiego drzewa, że jeżeli D i D' są izomorficzne, to $\sigma(D) = \sigma(D')$. Następnie po prostu porównujemy otrzymane sygnatury. Jak można zdefiniować taką sygnaturę?

Rozważmy najpierw prosty przykład porównywania ciągów liczb. Chcielibyśmy wiedzieć, czy dwa ciągi reprezentują ten sam zbiór wartości. Jedno z możliwych rozwiązań polega na posortowaniu ich i stwierdzeniu, czy są identyczne. Sortowanie to w tym wypadku właśnie wyznaczenie sygnatury.

Izomorfizm drzew (z wyróżnionym korzeniem) polega wyłącznie na permutowaniu synów każdego węzła. Sygnaturę wyznaczymy zatem, biorąc jakąś wyróżnioną permutację. Jaką? Np. najmniejszą, o ile nauczymy się porównywać drzewa. Przyjmijmy taką definicję:

- jeżeli korzeń drzewa D ma mniej synów, niż korzeń drzewa D', to $D < D'$
- jeżeli korzeń r drzewa D oraz korzeń r' drzewa D' mają po n synów, oraz $D_1, D_2, \ldots, D_n$ jest ciągiem poddrzew drzewa D, zakorzenionych w kolejnych synach r, analogicznie $D'_1, \ldots, D'_n$, oraz $D_1 = D'_1, D_2 = D'_2, \ldots, D_{k-1} = D'_{k-1}$ i przy tym $D_k < D'_k$ dla pewnego $k \leq n$, to także $D < D'$ (tzn. aby porównać drzewa D i D', potrzebujemy porównać leksykograficznie ciągi synów ich korzeni).

Chwili namysłu wymaga poprawność tej definicji. Niech każdy Czytelnik, który widzi ją po raz pierwszy, sam przeanalizuje, dlaczego z każdych dwóch nieidentycznych drzew jedno okazuje się być mniejsze od drugiego. Warto też sprawdzić, że jeżeli $D_1 < D_2$ oraz $D_2 < D_3$, to $D_1 < D_3$.

Za sygnaturę drzewa obieramy teraz po prostu najmniejsze możliwe w sensie określonego wyżej porządku drzewo, izomorficzne z danym. Wyznaczamy ją w ten sposób, że idąc od liści do korzenia, w każdym wierzchołku sortujemy synów w kolejności od tego, pod którym jest zaczepione najmniejsze poddrzewo (w zdefiniowanym wyżej sensie) do tego, pod którym zaczepione jest największe.

Sądzę, że nie jest już teraz trudno zdefiniować sygnatury dla zadania o pchłach. Nasze grafy rozpadają się na rozłączne cykle, z których do każdego podczepione są drzewa. Pierwszym krokiem jest zastąpienie każdego drzewa jego sygnaturą. Następnie można znaleźć dla każdego cyklu taką jego reprezentację (rotację) $(v_1, \ldots, v_n)$, by po zastąpieniu każdego wierzchołka w tym ciągu sygnaturą podczepionego do niego drzewa (być może zbudowanego tylko z korzenia), otrzymać leksykograficznie najmniejszy możliwy ciąg drzew. Z kolei można porównywać tak otrzymane sygnatury różnych cykli z podoczepianymi drzewami. W rozwiązaniu wzorcowym są one porównywane leksykograficznie. Inną możliwością byłoby najpierw porównywanie długości cykli, a następnie porównywanie leksykograficzne podczepionych drzew tylko dla cykli równej długości (analogicznie do definicji porządku na drzewach, podanej powyżej). Sygnaturą dla całego grafu jest uporządkowany ciąg wszystkich występujących w nim cykli. Sprawdzenie, że identyczne sygnatury otrzymujemy wtedy i tylko wtedy, gdy grafy są izomorficzne, nie powinno być problemem, jeśli ktoś potrafi udowodnić to dla drzew.

Jeżeli komuś powyższy opis nie wystarczył, to po szczegóły odsyłam do kodu programu wzorcowego.

Analiza złożoności rozwiązania

Mamy już jasność, że nasze zadanie da się rozwiązać w czasie wielomianowym. Spróbujmy dokładniej oszacować złożoność naszkicowanego algorytmu.

Wydzielenie w grafie cykli oraz wierzchołków nie leżących na cyklach i zbudowanie struktury ojców/synów w drzewach jest proste i może być wykonane w czasie liniowym.

Następnie dla każdego drzewa musimy obliczyć sygnaturę. Oznacza to konieczność posortowania synów każdego węzła. Sortowanie wymaga wykonania pewnej liczby porównań, a każde porównanie w pesymistycznym przypadku kosztuje tyle, co minimum z rozmiaru porównywanych poddrzew. W rozwiązaniu firmowym dla uproszczenia zastosowano sortowanie przez wstawianie. W sortowaniu tym każda para elementów jest porównywana co najwyżej raz. Pozwala to oszacować koszt wyznaczenia sygnatury całego drzewa poprzez następującą obserwację: każdy węzeł drzewa co najwyżej raz bierze udział w porównaniu z poddrzewem, zawieszonym w każdym innym węźle, leżącym na takiej jak on lub mniejszej głębokości (odległości od korzenia). Jeżeli drzewo ma n wierzchołków, pozwala to oszacować koszt obliczenia sygnatury tego drzewa przez $O(n^2)$ (gdyż w sumie we wszystkich operacjach porównania poddrzew każdy z wierzchołków będzie brał udział co najwyżej n razy). Oczywiście, jeżeli mamy las drzew, które w sumie mają n wierzchołków, to tym bardziej łączny koszt wyznaczenia sygnatury dla każdego z nich jest $O(n^2)$.

Z kolei trzeba wyznaczyć sygnatury dla poszczególnych cykli. Dla cyklu długości k wymaga to k porównań cykli (a dokładniej, różnych rotacji tego samego cyklu), by znaleźć minimalną rotację. Każde porównanie angażuje każdy z wierzchołków grafu co najwyżej raz, więc w sumie znalezienie minimalnej rotacji dla każdego z cykli wymaga $O(n^2)$ operacji.

Na koniec podobna argumentacja co przy wyznaczaniu sygnatur pozwala stwierdzić, że posortowanie przez wstawianie wszystkich cykli w grafie o n wierzchołkach również wymaga $O(n^2)$ operacji. Podsumowując, całe zadanie rozwiążemy w czasie $O(n^2)$.

Dla szczególnie dociekliwego Czytelnika mamy zadanie: czy można ten problem rozwiązać w czasie mniejszym od kwadratowego? Dla których etapów obliczenia (wyznaczanie sygnatur drzew, cykli, całego grafu) potrafiłbyś znaleźć szybszy algorytm?

Testy

Ponieważ zadanie wymaga podania jedynie odpowiedzi „tak” lub „nie”, więc każdy z 10 właściwych testów obejmował od 20 do 100 przypadków, aby wyeliminować programy, próbujące zgadywać odpowiedź na chybił–trafił.
Testy zostały wygenerowane losowo, przy użyciu następujących parametrów:

- d — liczba przypadków
- n_{min}, n_{max} — ograniczenia na liczbę żetonów
- c_{min}, c_{max} — ograniczenia na liczbę cykli
- t_{min}, t_{max} — ograniczenia na liczbę drzew, doczepionych do każdego z cykli
- α_{min}, α_{max} — ograniczenia na to, jaki ułamek liczby żetonów w każdej ze składowych grafu znajduje się na cyklu
- p_{min}, p_{max} — ograniczenia na współczynnik, sterujący stopniem węzłów drzew (im wyższy współczynnik, tym większa statystycznie liczba synów w każdym węźle, a zatem drzewa szersze i niższe)

Wartości parametrów dla poszczególnych testów przedstawia tabela:

Nr	d	n_{min}	n_{max}	c_{min}	c_{max}	t_{min}	t_{max}	α_{min}	α_{max}	p_{min}	p_{max}
0	2	test z treści zadania									
1	100	4	5	1	5	1	5	0,01%	100%	0,0%	100%
2	100	9	10	1	10	1	10	0,01%	100%	0,0%	100%
3	100	19	20	1	20	1	20	0,01%	100%	0,0%	100%
4	100	90	100	1	100	1	100	0,01%	100%	0,0%	100%
5	100	90	100	1	10	1	5	0,01%	100%	0,0%	100%
6	100	90	100	1	10	1	5	0,01%	50%	0,0%	100%
7	100	90	100	1	10	1	5	0,01%	20%	0,1%	10%
8	40	400	500	1	500	1	500	0,01%	100%	0,0%	100%
9	5	1900	2000	500	2000	1	2000	0,01%	100%	0,0%	100%
	5	1900	2000	1	2	1	1	0,01%	5%	0,003%	0,03%
	5	1900	2000	1	2	1	1	0,01%	5%	96,7%	100%
	5	1900	2000	10	20	1	5	0,01%	20%	0,1%	1%
10	5	1900	2000	1	2	1	1	0,01%	5%	0,003%	0,03%
	5	1900	2000	1	2	1	1	0,01%	5%	96,7%	100%
	5	1900	2000	500	2000	1	2000	0,01%	100%	0,0%	100%
	5	1900	2000	10	20	1	5	0,01%	20%	0,1%	1%

Testy 1–3 były prostymi testami poprawnościowymi. Niewykluczone, że mogły być rozwiązane nawet przez algorytm wykładniczy, który np. szukałby izomorfizmu, badając wszystkie możliwe permutacje zbioru wierzchołków grafu. Na testach 4–7 pewne szanse miały rozwiązania, działające w czasie sześciennym. Przez ostatnie trzy testy przechodziły tylko algorytmy o złożoności czasowej $O(n^2)$.

Porównywanie naszyjników

W Bajtocji żyje bardzo znany jubiler Bajtazar. Zajmuje się on wyrobem naszyjników. Naszyjniki są zrobione z drogocennych kamieni nanizanych na nitkę. Do wyrobu naszyjników używa się 26-ciu rodzajów kamieni, będziemy je oznaczać małymi literami alfabetu (angielskiego): `a`, `b`, ..., `z`. *Bajtazar postawił sobie za punkt honoru, aby nigdy nie wykonać dwóch takich samych naszyjników i przechowuje opisy wykonanych przez siebie naszyjników. Niektóre z tych naszyjników są bardzo długie. Dlatego też ich opisy mają skróconą postać. Każdy opis składa się z szeregu wielokrotnie powtórzonych sekwencji kamieni (wzorów). Opis naszyjnika to ciąg wzorów wraz z liczbami ich powtórzeń. Każdy wzór jest opisany za pomocą sekwencji liter reprezentujących kamienie tworzące wzór. Przykładowo, opis:* `abc` *2* `xyz` *1* `axc` *3 reprezentuje naszyjnik* `abcabcxyzaxcaxcaxc` *powstały przez dwukrotne powtórzenie wzoru* `abc`, *jednokrotne wystąpienie wzoru* `xyz` *i trzykrotne powtórzenie wzoru* `axc`. *Sprawę dodatkowo utrudnia fakt, iż naszyjniki nie mają widocznego początku, ani końca i można je dowolnie obracać w kółko. Powyższy opis reprezentuje również np. naszyjniki* `cabcxyzaxcaxcaxcab` *oraz* `xcaxcaxcabcabcxyza`.

Zadanie

Napisz program, który:

- *wczyta z pliku wejściowego* `nas.in` *dwa opisy naszyjników,*
- *sprawdzi, czy opisy te reprezentują takie same naszyjniki,*
- *zapisze wynik do pliku* `nas.out`.

Wejście

W pierwszym i drugim wierszu pliku tekstowego `nas.in` *znajdują się opisy naszyjników, po jednym w wierszu. Każdy z nich składa się z sekwencji liczb całkowitych i słów złożonych z małych liter alfabetu angielskiego, pooddzielanych pojedynczymi odstępami. Opis naszyjnika składa się z liczby całkowitej n równej liczbie wzorów występujących w opisie naszyjnika ($1 \leq n \leq 1\,000$), po której występuje n opisów powtórzeń wzorów. Opis powtórzeń i-tego wzoru składa się z: liczby całkowitej l_i równej długości wzoru ($1 \leq l_i \leq 10\,000$), słowa s_i złożonego z l_i małych liter alfabetu (angielskiego)* `a`, ..., `z`, *reprezentującego wzór oraz liczby całkowitej k_i równej liczbie powtórzeń wzoru s_i ($1 \leq k_i \leq 100\,000$). Wiadomo, że suma liczb l_i (dla $i = 1, \ldots, n$) nie przekracza $10\,000$.*

Wyjście

Twój program powinien zapisać, w pierwszym i jedynym wierszu pliku wyjściowego `nas.out`, *słowo "TAK", jeśli obydwa opisy przedstawiają taki sam naszyjnik, lub słowo "NIE", w przeciwnym przypadku.*

Przykład

Dla pliku wejściowego `nas.in`*:*

```
3 3 abc 2 3 xyz 1 3 axc 3
4 4 cabc 1 4 xyza 1 3 xca 3 1 b 1
```

poprawną odpowiedzią jest plik wyjściowy `nas.out`*:*

```
TAK
```

Zmiany w treści zadania

Podczas zawodów dokonano następującej zmiany w treści zadania:

Naszyjniki powstałe jeden z drugiego przez odwrócenie kolejności kamieni nie muszą być identyczne, a więc np. opisy 'abc' i 'cba' nie reprezentują tego samego naszyjnika.

Rozwiązanie

Zadanie sprowadza się do sprawdzenia, czy dwa dane słowa (ciągi znaków) są *cyklicznie równoważne*, to znaczy, czy jedno można otrzymać z drugiego przez jego cykliczne przesunięcie. Nietrudno zauważyć, że mające takie same długości słowa u i w są cyklicznie równoważne wtedy i tylko wtedy, gdy u występuje jako podsłowo słowa $w \cdot w$ (gdzie $\cdot$ oznacza konkatenację, czyli sklejenie słów). Nasz problem można by zatem rozwiązać w czasie proporcjonalnym do długości badanych słów, stosując efektywny algorytm wyszukiwania wzorca w tekście. Znane są dość wyrafinowane algorytmy wyszukiwania wzorca w czasie liniowym bez użycia pomocniczych tablic (zob. np. [?]), jednak cykliczną równoważność można sprawdzić znacznie prościej (zob. [?]):

Wprowadźmy oznaczenia:

- $|a|$ to długość słowa a;
- $u^{(k)} = u[k+1..n] \cdot u[1..k]$ (cykliczne przesunięcie słowa u o k pozycji w lewo);
- $D1 = \{1 \leq k \leq n : w^{(k-1)} >_L u^{(j)}$ dla pewnego $j\}$, gdzie $>_L$ oznacza porządek leksykograficzny na słowach;
- podobnie $D2 = \{1 \leq k \leq n : u^{(k-1)} >_L w^{(j)}$ dla pewnego $j\}$.

```
1: { Algorytm sprawdzania, czy u powstaje przez cykliczne przesunięcie w }
2: { Niech x = uu$, y = ww#, n=|| u|| }
3: begin
4:     i := 0; j := 0; k := 1;
5:     while (i < n) and (j < n) and (k ≤ n) do
6:     begin
7:         k := 1;
8:         while x[i + k] = y[j + k] do
9:             k := k + 1;
10:        if k ≤ n then
11:            if x[i + k] > y[j + k] then
12:                i := i + k
13:            else
14:                j := j + k
15:    { Niezmiennik: [1..i] ⊆ D1, [1..j] ⊆ D2 }
16:    end
17: end;
18: { u jest cyklicznym przesunięciem w wtedy i tylko wtedy, gdy k > n }
```

Algorytm działa w czasie liniowym względem n, a jego poprawność wynika z podanego niezmiennika oraz z faktu, że jeśli $D1 = [1..n]$ lub $D2 = [1..n]$, to słowa u i w nie są cyklicznie równoważne.

Dodatkową trudność w zadaniu stanowi to, że opisy naszyjników są podane w formie skompresowanej. Żeby uzyskać program działający w czasie proporcjonalnym nie do faktycznego rozmiaru samych naszyjników, ale raczej do rozmiaru ich *opisów*, musimy odpowiednio zaimplementować pętlę w wierszach 8–9. W naszym przypadku wystarczy, żebyśmy potrafili efektywnie rozstrzygać, czy któreś ze słów a^l, b^r (gdzie a^l oznacza konkatenację l kopii słowa a) jest prefiksem (czyli fragmentem początkowym) drugiego słowa. Nietrudno pokazać, że jeśli $|a^l|, |b^r| \geq |a| + |b|$, to powyższy warunek jest równoważny temu, że $a \cdot b = b \cdot a$. Stąd wynika, że w celu stwierdzenia, czy któreś ze słów a^l, b^r jest prefiksem drugiego, wystarczy porównać tylko $|a| + |b|$ początkowych liter tych słów.

Usprawnienia wymagają jeszcze wiersze 12 i 14 w algorytmie. Z podanego niezmiennika wynika, że jeśli na pozycji $i + k$ (odpowiednio $j + k$) mamy do czynienia z co najmniej drugim powtórzeniem pewnego wzoru, to bez zaburzenia niezmiennika możemy od razu przeskoczyć do ostatniego powtórzenia tego wzoru.

Testy

Do sprawdzania rozwiązań zawodników użyto 11 grup testów (po 3 testy w każdej grupie). Oto ich krótka charakterystyka:

- małe testy poprawnościowe (grupy 1–3)

- średnie testy poprawnościowe (grupy 4–5): 5–10 wzorów długości około 100, powtarzających się około 500 razy
- duże testy wydajnościowe (grupy 6–8): wzory długości 1000–3000, powtarzające się około 10000 razy
- bardzo duże testy wydajnościowe (grupy 9–11): suma długości wzorów 5000–10000, 50000–100000 powtórzeń.

Krzysztof Onak
Treść zadania, Opracowanie

Tomasz Waleń
Program

Zwiedzanie miasta

Bajtocka Agencja Turystyczna (w skrócie BAT) chce wejść na rynek oferując zwiedzanie Bajtogrodu autobusem–kabrioletem. Należy zbudować siedzibę firmy, w której będzie się zaczynało i kończyło zwiedzanie. Trasa zwiedzania musi przechodzić wszystkimi ulicami miasta, w przeciwnym przypadku turyści mogliby podejrzewać, że nie zobaczyli czegoś bardzo interesującego.

Ulice nie muszą być proste i mogą przebiegać tunelami lub wiaduktami. Wszystkie ulice są dwukierunkowe. Każda ulica łączy dwa skrzyżowania. Z każdego skrzyżowania w czterech kierunkach wychodzą ulice. Może się zdarzyć, że dwa skrzyżowania są połączone więcej niż jedną ulicą. Na ulicach nie wolno zawracać, ale można to robić na skrzyżowaniach. Ponadto wiadomo, że z każdego skrzyżowania da się dojechać do każdego innego.

Przy każdej ulicy, dokładnie w połowie drogi pomiędzy skrzyżowaniami, które łączy ulica, znajduje się szczególnie godna podziwu atrakcja turystyczna (np. piękny widok, pomnik lub inny zabytek), wywierająca na zwiedzających `wrażenie" określone nieujemną liczbą całkowitą. Siedziba BATu powinna znajdować się przy jednej z takich atrakcji.

Przy doborze trasy zwiedzania należy brać pod uwagę zainteresowanie turystów, które może się zmieniać w trakcie zwiedzania. Przejechanie autobusem jednej bajtomili powoduje spadek zainteresowania o jeden. Przejechanie **po raz pierwszy** *obok danej atrakcji turystycznej zwiększa zainteresowanie turystów, o liczbę określająca wrażenie, jakie robi atrakcja. Początkowo poziom zainteresowania turystów jest równy wrażeniu, jakie robi atrakcja, przy której znajduje się siedziba BATu. Zainteresowanie turystów nie może w trakcie wycieczki nigdy spaść poniżej zera.*

Zadanie

Napisz program, który:

- *wczyta opis miasta z pliku tekstowego* `zwi.in`,
- *znajdzie trasę spełniającą podane wymagania, lub stwierdzi, że taka trasa nie istnieje,*
- *zapisze wynik do pliku tekstowego* `zwi.out`.

Wejście

W pierwszym wierszu pliku tekstowego `zwi.in` *znajduje się jedna liczba całkowita n określająca liczbę skrzyżowań, $1 < n \leq 10\ 000$. Skrzyżowania są ponumerowane od 1 do n, a ulice są ponumerowane od 1 do $2n$. Kolejnych $2n$ wierszy opisuje ulice — $(i+1)$-szy wiersz w pliku opisuje ulicę o numerze i. W każdym wierszu znajdują się cztery liczby całkowite a, b, l, s oddzielone pojedynczymi odstępami. Liczby a i b to numery skrzyżowań, które łączy dana ulica, $1 \leq a, b \leq n$, $a \neq b$. Liczba l jest parzystą liczbą całkowitą będącą długością ulicy w bajtomilach, $2 \leq l \leq 1\ 000$. Atrakcja turystyczna położona przy danej ulicy robi wrażenie określone liczbą s, $0 \leq s \leq 1\ 000$.*

Wyjście

Pierwszy wiersz pliku tekstowego `zwi.out` *powinien zawierać jedno słowo* `TAK`*, jeżeli istnieje taka trasa, lub* `NIE`*, w przeciwnym przypadku. Jeśli odpowiedź jest pozytywna to kolejne wiersze powinny opisywać przykładową trasę. Drugi wiersz powinien zawierać dokładnie jedną liczbę całkowitą k równą liczbie skrzyżowań występujących na trasie zwiedzania. (Pamiętaj, że ulica, przy której ma znajdować się siedziba BATu łączy pierwsze i ostatnie skrzyżowanie). Oznaczmy przez s_i (dla $i = 1, 2, \ldots, k$) numer ulicy, którą podczas zwiedzania dojeżdża się do i-tego (w kolejności zwiedzania) skrzyżowania. Kolejny wiersz powinien zawierać dwie liczby całkowite s_1 i d równe odpowiednio numerowi ulicy, przy której należy zbudować siedzibę BATu oraz numerowi pierwszego skrzyżowania, przez które prowadzi trasa zwiedzania. Kolejne $k-1$ wierszy powinno zawierać po jednej liczbie całkowitej, odpowiednio s_2, s_3, …, s_k.*

Przykład

Dla pliku wejściowego `zwi.in`*:*

```
4
1 2 4 6
2 4 2 4
```

```
3 2 4 2
4 3 10 8
2 1 8 7
4 3 2 1
1 4 2 6
3 1 4 5
```

poprawną odpowiedzią jest plik wyjściowy `zwi.out`*:*

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

Rozwiązanie

W grafie nieskierowanym definiujemy *cykl Eulera* jako cykl przechodzący przez każdą krawędź grafu dokładnie raz. Rozpatrzmy graf, w którym wierzchołkami będą skrzyżowania, a krawędziami ulice. Nietrudno dostrzec, że istnieje w tym grafie cykl Eulera, ponieważ każdy wierzchołek ma parzysty stopień — z każdego skrzyżowania wychodzą cztery ulice. Znalezienie pewnej trasy przebiegającej po takim cyklu wydaje się być dobrym pomysłem, gdyż agencja turystyczna „traci" wówczas najmniej z zadowolenia turystów. Tymczasem mamy:

Fakt 1 *W grafie opisanym przez poprawne dane wejściowe istnieje cykl Eulera.*

Weźmy teraz pewien cykl o długości k, gdzie $k > 2$, z wierzchołkami ponumerowanymi kolejno od 1 do k. Przypiszmy każdemu wierzchołkowi pewną nieujemną liczbę — wierzchołkowi nr i przyporządkowujemy liczbę w_i. Niech l_i będzie długością krawędzi od wierzchołka nr i do $i+1$, jeśli $i < k$, albo do 1 w przeciwnym przypadku. Rozważamy teraz następującą sytuację: wybieramy pewien wierzchołek i obchodzimy cykl w kierunku zgodnym z numeracją $1 \to 2 \to \dots k \to 1$, wracając do punktu wyjściowego. Przypuśćmy, że przejście krawędzi kosztuje tyle, co jej długość, a w każdym wierzchołku otrzymujemy zwrot kosztu w_i dla tego wierzchołka. Niech $B = \sum_{i=1}^{k}(w_i - l_i)$ i przyjmijmy, że na początku dysponujemy kapitałem równym 0. Wówczas prawdziwe jest następujące stwierdzenie:

Fakt 2 *Cykl można obejść zachowując zawsze nieujemne konto wtedy i tylko wtedy, gdy* $B \geq 0$.

Dowód: Przypuśćmy najpierw, że cykl można obejść w ten sposób. Wtedy bilans podróży wyrażający się jako B musi być nieujemny. Teraz dowód w drugą stronę. Zakładamy, że $B \geq 0$. Rozpocznijmy symulację w wierzchołku o numerze 1. Przechodzimy cykl przy założeniu, że możemy mieć ujemny stan konta. Oznaczamy przez b_i stan konta już po dojściu do wierzchołka numer i, ale jeszcze przed pobraniem wyznaczonej rekompensaty. Na starcie mamy $b_1 = 0$. Dla pewnego j otrzymujemy minimalną wartość b_j. Wierzchołek o numerze j będzie naszym nowym wierzchołkiem startowym. Rozpoczynając drogę tym razem w j dostajemy nowe wartości b'_i takie, że:

$$b'_i = \begin{cases} b_i - b_j + B & \text{dla } i < j, \\ 0 & \text{dla } i = j, \\ b_i - b_j & \text{dla } i > j. \end{cases}$$

Widać już teraz, że b'_i nie może być ujemne dla żadnego i, gdyż dla każdego i zachodzi $b_i \geq b_j$, czyli $b_i - b_j \geq 0$.

Wracamy teraz do naszego zadania. Fakt 1 gwarantuje nam istnienie cyklu Eulera. W dowolnie znalezionym cyklu Eulera jako wierzchołki traktujemy teraz atrakcje turystyczne. Fakt 2 daje nam ostatecznie, że rozwiązanie zadania istnieje wtedy i tylko wtedy, gdy suma wrażeń dostarczanych przez atrakcje jest nie mniejsza od sumy długości ulic, a ponadto do znalezienia takiego rozwiązania można posłużyć się dowolnym cyklem Eulera. Możemy już zapisać główne kroki algorytmu rozwiązującego zadanie:

1. wczytaj dane;
2. sprawdź, czy wrażeń dostarczanych przez atrakcje turystyczne wystarczy na zwiedzenie miasta, a jeśli tak to:

   (a) znajdź cykl Eulera,
   (b) znajdź odpowiedni punkt startowy na skonstruowanym cyklu;

3. zapisz wynik.

Punkt 2(a) można zrealizować przechodząc graf w głąb po nieodwiedzonych jeszcze krawędziach. Odpowiednia procedura może mieć następujący pseudokod:

```
procedure euler( v);
begin
  for w in Sąsiedzi( v) do
    if not odwiedzona[ v–w] then
      begin
        odwiedzona[ v–w]:=true;
        euler( w);
        DopiszKrawędź( v–w);
       { dopisuje krawędź na koniec początkowo pustej listy }
      end
end
```

Podaną procedurę wywołujemy dla dowolnego wierzchołka, na przykład dla $v = 1$. Nie trudno dowieść, że znajduje ona cykl Eulera. Przy sprawnym zaimplementowaniu ten krok algorytmu wymaga czasu $O(n)$.

W implementacji punktu 2(b) algorytmu, można posłużyć się konstruktywnym dowodem faktu 2. Ta część algorytmu działa również w czasie $O(n)$. Podsumowując, cały algorytm działa w czasie $O(n)$.

Testy

Do oceny rozwiązań zawodników użyto kompletu 12 testów:

- `zwi1a.in` — prosty test poprawnościowy, $n = 4$;
- `zwi1b.in` — prosty test poprawnościowy na odpowiedź NIE, $n = 4$;
- `zwi2a.in` — graf pełny, $n = 5$;
- `zwi2b.in` — prosty test poprawnościowy, $n = 10$;
- `zwi3.in` — losowy graf, $n = 100$;
- `zwi4.in` — okrąg z losowymi wagami, $n = 1000$;
- `zwi5.in` — dwa nałożone na siebie losowe cykle, $n = 1000$;
- `zwi6.in` — drabinka, $n = 3000$;
- `zwi7.in` — dwa nałożone na siebie losowe cykle, $n = 5000$;
- `zwi8.in` — dwa równoległe okręgi połączone krawędziami, $n = 6000$;
- `zwi9.in` — graf losowy, $n = 10000$;
- `zwi10.in` — dwa nałożone na siebie losowe cykle, $n = 10000$.

Testy `zwi1a.in` i `zwi1b.in` oraz `zwi2a.in` i `zwi2b.in` były zgrupowane. Różnice pomiędzy sumą wrażeń i sumą długości ulic były w testach niewielkie, a krawędzie w plikach znajdowały się w losowej kolejności.

Marcin Kubica
Treść zadania, Opracowanie

Marcin Sawicki
Program

Bank

W Bajtocji funkcjonują cztery rodzaje waluty: denary, franki, grosze i talary, nie wymienialne między sobą. Przysparza to wiele kłopotów mieszkańcom Bajtocji.

Bajtocki Bank Biznesu (w skrócie BBB) na skutek pomyłki w rodzaju waluty stanął w obliczu utraty płynności gotówkowej. Zawarł on z klientami szereg umów na kredytowanie różnych przedsięwzięć. Wszystkie te umowy są zawarte według takiego samego wzoru:

- *umowa określa maksymalną wysokość kredytu w każdym rodzaju waluty,*
- *w ramach tak określonego limitu, gdy klient potrzebuje gotówki, to zgłasza się do BBB prosząc o określoną sumę w każdym rodzaju waluty; BBB może dowolnie długo zwlekać z wypłaceniem pieniędzy, ale dopóki klient nie przekracza maksymalnej wysokości kredytu, to prędzej czy później musi je klientowi wypłacić,*
- *po otrzymaniu pieniędzy klient może zgłaszać się po kolejne transze, aż do wyczerpania limitu,*
- *na koniec klient spłaca całość zaciągniętego kredytu, może z tym zwlekać dowolnie długo, ale prędzej czy później musi spłacić kredyt,*
- *klient nie ma obowiązku wykorzystania kredytu w maksymalnej wysokości,*
- *dla uproszczenia zakładamy, że klienci nie płacą żadnych odsetek ani prowizji.*

BBB nie dysponuje wystarczającą ilością gotówki, aby zaspokoić potrzeby swoich klientów, a bez ich wcześniejszego zaspokojenia kredyty nie będą spłacane. BBB poprosił Bajtocki Fundusz Walutowy (w skrócie BFW) o pomoc. BFW zgodził się pomóc BBB, ale zażądał, żeby BBB określił minimalne kwoty każdego rodzaju waluty, jakie BBB musi posiadać, aby móc doprowadzić do spłacenia przez klientów wszystkich kredytów (nawet jeżeli klienci będą chcieli wykorzystać swoje kredyty do maksymalnej ich wysokości).

Specjaliści BBB odkryli, że możliwych jest wiele odpowiedzi na tak postawione pytanie (por. przykład). BFW odpowiedziało, że interesują ich dowolne takie kwoty poszczególnych rodzajów walut, że gdyby zmniejszyć którąkolwiek z nich choćby o 1, to mogłyby nie wystarczyć do zakończenia realizacji wszystkich kredytów.

Zadanie

Napisz program, który:

- *wczyta z pliku* `ban.in` *maksymalne i aktualne wysokości kredytów klientów,*
- *wyznaczy minimalne kwoty poszczególnych rodzajów walut gwarantujące możliwość realizacji wszystkich kredytów,*
- *zapisze wynik w pliku* `ban.out`.

Jeśli jest możliwych wiele wyników, to Twój program powinien zapisać dowolny z nich.

Wejście

W pierwszym wierszu pliku tekstowego `ban.in` *jest zapisana jedna dodatnia liczba całkowita n równa liczbie klientów, $1 \leq n \leq 8\,000$. Klienci są ponumerowani od 1 do n. W kolejnych n wierszach jest zapisanych po osiem nieujemnych liczb całkowitych. W $i+1$-szym wierszu (dla $i = 1, \ldots, n$) zapisane są liczby $m_{i,1}, m_{i,2}, m_{i,3}, m_{i,4}, w_{i,1}, w_{i,2}, w_{i,3}, w_{i,4}$, ($0 \leq w_{i,j} \leq m_{i,j} \leq 50\,000$, dla $j = 1, \ldots, 4$). Liczby $m_{i,j}$ i $w_{i,j}$ określają odpowiednio maksymalną i aktualną wysokość kredytu klienta nr i w: denarach ($j = 1$), frankach ($j = 2$), groszach ($j = 3$) i talarach ($j = 4$).*

Wyjście

Twój program powinien zapisać w pierwszym (i jedynym) wierszu pliku tekstowego `ban.out` *cztery nieujemne liczby całkowite, pooddzielane pojedynczymi odstępami, określające minimalne kwoty gotówki, jakie musi posiadać BBB, odpowiednio w denarach, frankach, groszach i talarach.*

Przykład

Dla pliku wejściowego `ban.in`*:*

```
4
3 2 1 2 0 2 0 1
2 4 1 8 1 2 1 1
3 2 0 3 1 0 0 1
3 0 1 2 1 0 0 1
```

poprawną odpowiedzią może być plik tekstowy `ban.out`*:*

```
1 2 0 7
```

lub:

```
2 0 1 4
```

Zakleszczenie i algorytm bankiera

Zadanie to jest związane ze zjawiskiem zakleszczenia oraz algorytmem bankiera używanym do unikania zakleszczenia (zobacz [?], p. 7.5.3). Zjawisko zakleszczenia występuje w systemach operacyjnych, w których *współbieżnie* (tzn. równocześnie) może być wykonywanych wiele *procesów* (działających programów). Procesy te mogą używać rozmaitych zasobów systemowych, takich jak pamięć, czy urządzenia wejścia/wyjścia. Zasoby te są przydzielane procesom przez system operacyjny. Zakleszczenie występuje wówczas, gdy kilka procesów czeka nawzajem na siebie, prosząc system o przydzielenie zasobów, które są zajęte przez pozostałe procesy. Wyobraźmy sobie na przykład, że w systemie jest jeden napęd CD–ROM i jedna karta dźwiękowa, oraz że dwa procesy (P_1 i P_2) chcą uzyskać wyłączny dostęp do tych urządzeń. Załóżmy przy tym, że P_1 uzyskał już dostęp do CD–ROM'u i czeka na zwolnienie karty dźwiękowej, a P_2 ma już dostęp do karty dźwiękowej i czeka na zwolnienie CD-ROM'u. Jak łatwo zauważyć, te dwa procesy będą na siebie czekać w nieskończoność. Takie zjawisko nazywamy zakleszczeniem. Oczywiście zakleszczenie jest zjawiskiem niepożądanym.

Jeden ze sposobów radzenia sobie z zakleszczeniem polega na takim przydzielaniu przez system operacyjny zasobów procesom, aby unikać zakleszczenia. Pomysł polega na tym, aby utrzymywać system w *bezpiecznym* stanie, tzn. takim stanie, w którym mamy gwarancje, że wszystkie działające procesy mogą zostać wykonane aż do końca, bez zakleszczenia. System operacyjny przydziela zasoby procesom tylko wtedy, gdy prowadzi to do bezpiecznego stanu. W rezultacie, może się zdarzyć, że ze względów bezpieczeństwa proces musi czekać, mimo że potrzebne mu zasoby są dostępne.

W celu stwierdzenia czy stan systemu jest bezpieczny używany jest *algorytm bankiera*. Algorytm ten potrzebuje dodatkowej informacji: maksymalnych ilości zasobów jakie mogą być potrzebne poszczególnym procesom. Procesy deklarują maksymalne ilości potrzebnych zasobów w momencie uruchomienia. Algorytm bankiera opiera się na analogii między systemem operacyjnym, a opisanym w treści zadania systemem bankowym. Różne rodzaje zasobów to różne, wzajemnie nie wymienialne między sobą waluty. Klienci to procesy działające w systemie, a BBB to system operacyjny. Środki pieniężne jakimi dysponuje BBB to wolne zasoby, a pieniądze pożyczone klientom to zasoby przydzielone procesom. Natomiast maksymalne ilości potrzebnych zasobów deklarowane przez procesy, to wysokości limitów określone w umowach kredytowych.

Algorytm bankiera opiera się na następujących obserwacjach. Jeżeli w danym stanie system może doprowadzić do zakończenia wszystkich procesów, to może to również zrobić wykonując i kończąc te procesy w pewnej kolejności, po jednym na raz. Aby móc zakończyć jakiś proces, musimy mieć w systemie tyle wolnych zasobów, żeby zaspokoić jego potrzeby. W najgorszym przypadku ilość potrzebnych zasobów jest równa różnicy między maksymalną zadeklarowaną liczbą potrzebnych zasobów, a ilością aktualnie przydzielonych zasobów. Jednak po zakończeniu procesu w systemie może tylko przybyć wolnych zasobów, gdyż wszystkie zasoby przydzielone procesowi są zwalniane. Tak więc, aby odpowiedzieć na pytanie czy sytuacja jest bezpieczna, musimy stwierdzić, czy istnieje taka kolejność $P_1, P_2, \ldots, P_n$, w której możemy wykonywać i kończyć procesy. Ponadto, taką kolejność możemy konstruować w sposób zachłanny — jeżeli istnieje taka kolejność i wolne w danej chwili zasoby wystarczają do zakończenia procesu P, to istnieje również taka kolejność zaczynająca się od P.

Niech n będzie liczbą procesów, a m liczbą rodzajów zasobów (w naszym przypadku $m = 4$). Zakładamy, że są określone następujące cztery tablice:

- *wolne* — wektor długości m określający ilości wolnych zasobów poszczególnych rodzajów,
- *maks* — macierz $n \times m$ określająca zadeklarowane przez poszczególne procesy maksymalne ilości potrzebnych im zasobów,
- *przydzielone* — macierz $n \times m$ określająca ilości zasobów przydzielonych poszczególnym procesom,

- *potrzebne* — macierz $n \times m$ określająca maksymalne ilości zasobów potrzebnych do zakończenia poszczególnych procesów; tę macierz możemy zawsze wyznaczyć na podstawie wzoru $potrzebne[i, j] = maks[i, j] - przydzielone[i, j]$.

Dodatkowo zakładamy, że mamy do dyspozycji dwa pomocnicze wektory: *pom* i *zakończone* długości odpowiednio m i n. Wektor *pom* reprezentuje symulowaną ilość wolnych zasobów, a *zakończone* reprezentuje zbiór procesów, które udało się zakończyć. Algorytm bankiera ma następującą postać:

```
pom := wolne;
zakończone := (false, false, ..., false);
while istnieje takie i, że:
not zakończone[i] and ∀j potrzebne[i, j] ≤ pom[j]
do begin
  for j := 1 to m do
     pom[j] := pom[j] + przydzielone[i, j];
   zakończone[i] := true
end;
if zakończone = (true, true, ..., true) then
  system jest w stanie bezpiecznym
else
  system nie jest w stanie bezpiecznym.
```

Algorytm ten ma złożoność $O(n^2 m)$.

Algorytm bankiera jest zwykle używany do określenia, czy stan systemu po przydzieleniu zasobów jest bezpieczny. W niniejszym zadaniu problem jest postawiony trochę inaczej. System znalazł się w stanie niebezpiecznym i pytanie dotyczy minimalnej liczby wolnych zasobów potrzebnych do tego, aby stan był bezpieczny.

Rozwiązanie

Możemy zastosować do rozwiązania tego zadania algorytm bankiera. Dla określonych danych wejściowych może istnieć wiele poprawnych wyników. Nasze rozwiązanie wyznacza wynik, który jest najmniejszy w porządku leksykograficznym. Inaczej mówiąc, wyznaczamy najpierw najmniejszą liczbę potrzebnych denarów, następnie dla tak określonej liczby denarów najmniejszą liczbę potrzebnych franków, itd. Liczba potrzebnych denarów jest z jednej strony nieujemna, a z drugiej nie przekracza sumy limitów na denary we wszystkich umowach kredytowych. Konkretną wartość znajdujemy w tym przedziale za pomocą metody bisekcji, stosując za każdym razem algorytm bankiera i sprawdzając czy przy danej liczbie denarów i nieograniczonych zasobach pozostałych walut stan jest bezpieczny — jeżeli nie, to liczba denarów jest zbyt mała. Jak wspomnieliśmy powyżej, koszt algorytmu bankiera wynosi $O(n^2 m)$. Jeżeli oznaczymy przez s maksymalną wysokość limitu jednej waluty w umowie kredytowej, to koszt wyznaczenia minimalnej liczby potrzebnych denarów wynosi $O(n^2 m \log(ns))$. Po ustaleniu liczy denarów możemy w podobny sposób wyznaczyć minimalną liczbę potrzebnych franków, zakładając, że mamy nieograniczone zasoby groszy i talarów. Podobnie wyznaczamy minimalną liczbę potrzebnych groszy oraz talarów. W ten sposób otrzymujemy rozwiązanie o złożoności rzędu $O(n^2 m^2 \log(ns))$.

Okazuje się, że możemy to zadanie rozwiązać bardziej efektywnie. Tak jak w powyższym algorytmie ustalamy minimalne potrzebne ilości kolejnych walut. Powiedzmy, że w kolejnym kroku wyznaczamy minimalną ilość k-tej waluty. Wykonujemy wówczas zmodyfikowany algorytm bankiera, który na podstawie ustalonych ilości walut 1, $\ldots$, $k-1$ i przy założeniu, że mamy nieograniczone zasoby walut $k+1, \ldots, m$, wyznacza minimalną potrzebną kwotę k-tej waluty. Symulujemy zakończenie kredytów w pewnej kolejności. W tym celu symulujemy pulę dostępnych środków pieniężnych oraz utrzymujemy zbiór kredytów, do zakończenia których mamy wystarczającą ilość walut $1, \ldots, k-1$. Symulując kończenie kredytów wybieramy za każdym razem taki kredyt, do zakończenia którego mamy wystarczająca ilość walut $1, \ldots, k-1$ i który wymaga najmniej środków k-tej waluty. Jeśli mamy wystarczającą ilość środków, to po prostu symulujemy zakończenie i spłatę tego kredytu. Jeśli natomiast brakuje środków k-tej waluty, to odpowiednio zwiększamy ich ilość. W ten sposób, po zakończeniu symulacji znamy minimalną wymaganą ilość środków waluty k.

Aby uzyskać dobrą złożoność takiego algorytmu, musimy zastosować odpowiednią strukturę danych do przechowywania puli kredytów, do zakończenia których mamy wystarczającą ilość walut $1, \ldots, k-1$. Używamy do tego celu stogów, uporządkowanych według kwoty określonej waluty potrzebnej do zakończenia kredytu. Zakładamy, że są zaimplementowane następujące operacje na stogach:

- *heap_init*(h) — inicjuje pusty stóg h,
- *heap_empty*(h) — określa czy stóg h jest pusty,

- *heap_put*(h, e, k) — wkłada na stóg h element e skojarzony z kluczem k,
- *heap_min*(h) — określa element o najmniejszym kluczu znajdujący się na stogu h,
- *heap_get_min*(h) — zdejmuje ze stogu element o najmniejszym kluczu i przekazuje go jako wynik.

Algorytm ten jest zaimplementowany przez poniższą procedurę. Zakładamy przy tym, że zadeklarowane są następujące zmienne:

```
const
   N_MAX = 8000;
   K_MAX = 4;
var
   wys_max, wys: array[1.. N_MAX, 1.. K_MAX] of word;
   n: word;
   akt: array [1.. K_MAX] of longint;
```

Zmienna n to liczba kredytów, a tablice *wys_max* i *wys* zawierają odpowiednio limity kredytów i aktualne zadłużenie klientów.

```
procedure oblicz;
var
   heaps: array[1.. K_MAX] of heap;
   rez: array[1.. K_MAX] of longint;
   i, j, k, e: word;
   zmiana: word;
begin
  for i := 1 to K_MAX do akt[ i] := 0;
  for k := 1 to K_MAX do { Ustalanie limitu na k-tą walutę. }
  begin
    for i := 1 to k do rez[ i] := akt[ i];
    for i := 1 to k do heap_init( heaps[ i]);
    for i := 1 to n do
       heap_put( heaps[1], i, wys_max[ i,1]- wys[ i,1]);
    for i := 1 to n do
    begin
      { Wybieramy kredyty mieszczące się w limitach na waluty 1..k-1. }
      for j := 1 to k-1 do
        while
          (not heap_empty( heaps[ j])) and
          ( heap_min( heaps[ j]) ≤ rez[ j])
        do begin
           e := heap_get_min( heaps[ j]);
           heap_put( heaps[ j+1], e, wys_max[ e, j+1]− wys[ e, j+1])
        end;
      { Element wymagający najmniej środków waluty k. }
       e := heap_get_min( heaps[ k]);
      { Czy trzeba zwiększyć akt[ k]? }
      if rez[ k] < wys_max[ e, k]− wys[ e, k] then
      begin
         zmiana := wys_max[ e, k]− wys[ e, k]− rez[ k];
        inc( rez[ k], zmiana);
        inc( akt[ k], zmiana)
      end;
      { Realizujemy kredyt. }
      for j := 1 to k do
        inc( rez[ j], wys[ e, j]);
    end;
  end;
end;
```

Testy

Testy zostały wygenerowane losowo, przy czym wysokość aktualnie wykorzystanego kredytu dla żadnej waluty nie przekracza 9. Poniższa tabelka ilustruje wielkości testów. Wszystkie testy można znaleźć na załączonej dyskietce.

nr	n	maksymalna kwota kredytu
1	6	13
2	8	15
3	10	20
4	20	100
5	100	1000
6	1000	4000
7	2000	10000
8	6000	50000
9	7000	50000
10	8000	50000

Tomasz Waleń
Treść zadania, Opracowanie

Marcin Sawicki
Program

Kopalnia złota

Bajtazar, zasłużony pracownik Bajtockiej Kopalni Złota, przechodzi w tym roku na emeryturę. W związku z tym, zarząd kopalni postanowił go uhonorować. W nagrodę za wieloletnią pracę, Bajtazar może otrzymać działkę — wycinek kopalni mający postać prostokąta o bokach równoległych do osi współrzędnych oraz szerokości s i wysokości w — położoną w dowolnie przez siebie wybranym miejscu. Oczywiście nie wszystkie lokalizacje działki są równie cenne. Wartość działki mierzy się liczbą samorodków złota znajdujących się na jej terenie (jeśli samorodek leży na granicy działki, to również znajduje się na jej terenie).

Twoim zadaniem jest napisanie programu umożliwiającego wyznaczenie jaką wartość ma najcenniejsza spośród wszystkich możliwych lokalizacji działek.

Dla uproszczenia przyjmujemy, że teren kopalni jest nieograniczony, jakkolwiek samorodki występują jedynie na ograniczonym obszarze.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `kop.in` *opis rozmieszczenia samorodków oraz wymiary działki,*
- *znajdzie wartość najcenniejszej spośród wszystkich lokalizacji działki, mierzoną liczbą znajdujących się na jej terenie samorodków,*
- *zapisze wynik w pliku tekstowym* `kop.out`*.*

Wejście

W pierwszym wierszu pliku tekstowego `kop.in` *zapisano dwie dodatnie liczby całkowite s i w oddzielone pojedynczym odstępem ($1 \leq s, w \leq 10\ 000$), oznaczające odpowiednio szerokość i wysokość działki. W drugim wierszu zapisano jedną dodatnią liczbę całkowitą n ($1 \leq n \leq 15\ 000$), oznaczającą liczbę samorodków znajdujących się na terenie kopalni. W kolejnych n wierszach zapisane są współrzędne poszczególnych samorodków. Każdy z tych wierszy zawiera dwie liczby całkowite x i y ($-30\ 000 \leq x, y \leq 30\ 000$), oddzielone pojedynczym odstępem, oznaczające odpowiednio współrzędną x i y samorodka.*

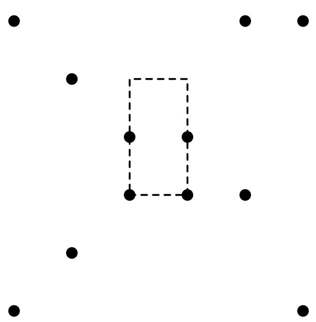
Wyjście

Plik tekstowy `kop.out` *powinien zawierać jedną liczbę całkowitą równą wartości najcenniejszej spośród wszystkich lokalizacji działek.*

Przykład

Dla pliku wejściowego `kop.in`*:*

```
1 2
12
0 0
1 1
2 2
3 3
4 5
5 5
4 2
1 4
0 5
5 0
2 3
3 2
```



poprawną odpowiedzią jest plik wyjściowy `kop.out`*:*

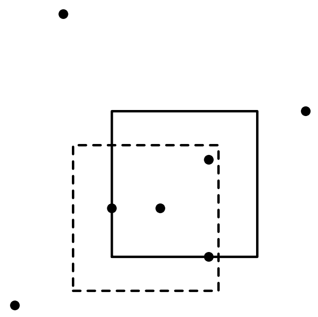
```
4
```

Rozwiązanie

Najprostszym rozwiązaniem tego zadania jest sprawdzenie wartości wszystkich możliwych działek, jednak z oczywistych powodów jest to rozwiązanie bardzo nieefektywne. Co prawda możemy ograniczyć się do takich działek, których wierzchołki pokrywają się z punktami kratowymi, ale nadal daje to $60\,000 \times 60\,000$ możliwości do sprawdzenia i co najgorsze, nawet w przypadku gdy ilość samorodków jest niewielka.

Do trochę lepszego rozwiązania prowadzi spostrzeżenie, iż zawsze możemy pokazać taką działkę o optymalnej wartości, na której lewym i dolnym brzegu leżą jakieś samorodki (każdą optymalną działkę można tak przesunąć by spełniała to kryterium, nie tracąc przy tym żadnych samorodków).

Rys. 1. Optymalna działka została zaznaczona przerywaną linią, a analogiczna działka po przesunięciu — linią ciągłą.



Zatem przestrzeń poszukiwań można już ograniczyć do $O(n^2)$ (na n różnych sposobów można wybrać samorodek na lewym brzegu i na n sposobów na dolnym brzegu), niestety nadal takie rozwiązanie będzie za mało efektywne w przypadku danych o dużych rozmiarach, ale dla danych o małych rozmiarach (np. prostych testów poprawnościowych) powinno być wystarczające.

Kod takiego rozwiązania może wyglądać następująco:

```
rozw := 0;
for i := 1 to n do
  for j := i to n do
  begin
     x := min(x_i,x_j);
     y := min(y_i,y_j);
     ile:= 0;
    { policzenie ile samorodków znajduje się w obrębie prostokąta }
    { (x,y)–(x+s,y+w) }
    for k := 1 to n do
      if (x ≤ x_i) and (y ≤ y_i) and
           (x_i ≤ x+s) and (y_i ≤ y+w) then inc( ile);
    if ile> rozw then rozw := ile
  end
```

Takie rozwiązanie ma złożoność $O(n^3)$, co przy maksymalnych danych jakie mogą wystąpić w zadaniu oznacza wykonanie około $3 \cdot 10^{12}$ operacji. Nawet przy założeniu, że dysponujemy szybkim komputerem, który potrafi wykonać 10^7 operacji w ciągu sekundy (jednak ta wartość jest bardzo uzależniona od wybranego języka programowania, kompilatora czy nawet metody kompilacji), a sam program przyspieszyć 3–krotnie (np. przez optymalizację), i tak oznaczałoby to, że program potrzebowałby na znalezienie rozwiązania 100 000s, czyli 27 godzin.

Jak więc ulepszyć poprzednie rozwiązanie? Można bardziej efektywnie zliczać liczbę samorodków w obrębie danego prostokąta, korzystając z bardziej zaawansowanych struktur danych, albo też można wykorzystać fakt, że wszystkie samorodki mamy dane na samym początku i analizować je wg. jakiegoś porządku.

W rozwiązaniu wzorcowym użyto popularnej techniki zwanej *zamiataniem*. Rozpatrzmy poziomą miotłę zamiatającą płaszczyznę od dołu do góry. Dla danego położenia miotły będziemy chcieli szybko obliczać wartości działek, których górny brzeg pokrywa się z aktualnym położeniem miotły. Podczas zamiatania płaszczyzny, dla każdego samorodka na miotle zaznaczamy przedział, w którym umieszczenie prawego górnego rogu działki obejmuje ten samorodek. Konieczne jest również dodanie sztucznych "ujemnych" samorodków, które będą służyły do usuwania samorodka (gdy miotła oddali się na ogległość w).

Szkic takiego rozwiązania wygląda następująco:

```
S := { (x_i, y_i, +1) oraz (x_i, y_i + w + 1, −1): dla 1 ≤ i ≤ n };
rozw := 0;
przeglądaj S w kolejności rosnących współrzędnych y
    (x,y,z) := { kolejny element z S };
    if z= +1 then
        dodajPrzedział( x, x+s)
    else
        usunPrzedział( x, x+s);
    rozw := max( rozw, MaksymalnyPrzedział);
```

Funkcja *MaksymalnyPrzedzial* szuka punktu na osi X, który należy do największej liczby przedziałów i zwraca ich liczbę.
Dla danych z przykładu podanego w treści zadania zbiór S wygląda następująco:

$$(0,0,+1),\ (0,3,-1),\ (2,2,+1),\ (2,5,-1),\ (3,3,+1),\ (3,6,-1),\ (4,5,+1),\ (4,8,-1),\ (5,5,+1),\ (5,8,-1),\ (4,2,+1),\ (4,5,-1),\ (1,4,+1),\ (1,7,-1),\ (0,5,+1),\ (0,8,-1),\ (5,0,+1),\ (5,3,-1),\ (2,3,+1),\ (2,6,-1),\ (3,2,+1),\ (3,5,-1).$$

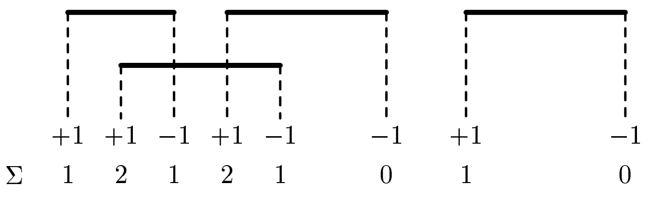
Zbiór S przeglądamy zgodnie z rosnącą współrzędną y, jednak wszystkie dla tej samej współrzędnej y trójki ze znakiem -1 powinny znaleźć się przed tymi ze znakiem $+1$.
W naszym wypadku mogłaby to być kolejność:

$$(0,0,+1),\ (5,0,+1),\ (2,2,+1),\ (4,2,+1),\ (3,2,+1),\ (5,3,-1),\ (0,3,-1),\ (3,3,+1),\ (2,3,+1),\ (1,4,+1),\ (3,5,-1).\ (4,5,-1),\ (2,5,-1),\ (4,5,+1),\ (5,5,+1),\ (0,5,+1),\ (3,6,-1),\ (2,6,-1),\ (1,7,-1),\ (4,8,-1),\ (5,8,-1),\ (0,8,-1).$$

Teraz pozostaje ustalić szczegóły: jakiej struktury danych użyć do reprezentowania przedziałów na miotle i jak szybko zrealizować funkcję *MaksymalnyPrzedział*?

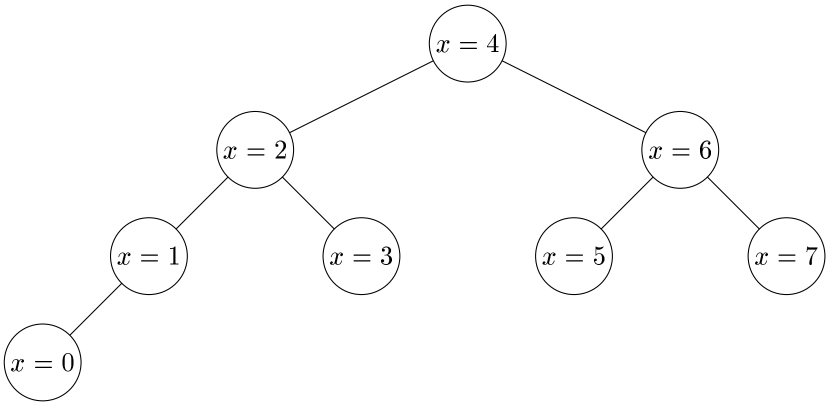
Odpowiedź na drugie pytanie można sprowadzić do problemu liczenia sum prefiksowych. Otóż gdy będziemy dodawać przedział $[a, b]$, to należy na osi X na pozycji a dodać $+1$, a na pozycji b dodać -1. Gdy będziemy chcieli usunąć jakiś przedział należy postąpić odwrotnie: na pozycji a dodać -1, a na b dodać $+1$. Teraz aby sprawdzić do ilu przedziałów należy dany punkt p, wystarczy zsumować wartości od minimalnej wartości na osi X do p, natomiast obliczenie funkcji *MaksymalnyPrzedzial* odpowiada znalezieniu maksymalnej sumy prefiksowej. Metoda ta została zilustrowana na rysunku 2.

Rys. 2. Przykład zastosowania sum prefiksowych do wyznaczania "maksymalnego" przedziału.



Bardzo często do reprezentowania miotły używa się drzew zrównoważonych takich jak AVL lub drzewa czerwono–czarne. Tak można postąpić i w tym wypadku, ale istnieje prostsze rozwiązanie. Ponieważ wszystkie punkty znamy zawczasu, możemy zbudować zrównoważone drzewo BST zawierające wszystkie wartości. Takie postępowanie może wydawać się trochę rozrzutne — w drzewach AVL przetrzymujemy tylko te wartości, które aktualnie są przetwarzane, a tu wszystkie. Jednak w tym wypadku punktów wcale nie jest aż tak dużo. Dodatkowo możemy takie drzewo ukryć w tablicy, dokładnie tak jak w przypadku kopców, a co za tym idzie, uniknąć dodatkowego kosztu związanego z przechowywaniem wskaźników, które definiują strukturę drzewa. Rysunek 3 pokazuje drzewo utworzone dla danych podanych w przykładzie zamieszczonym w treści zadania.

Rys. 3. Struktura miotły dla danych z przykładu podanego w treści zadania.



W każdym węźle drzewa oprócz wartości współrzędnej x (klucza) będziemy utrzymywać wartości:

- `MiotlaSuma` — oznaczającą sumę wartości dla kluczy z całego poddrzewa danego węzła (łącznie z nim samym),
- `MiotlaMaxSuma` — oznaczającą maksimum z wszystkich sum prefiksowych ciągu wartości zapisanych w poddrzewie danego węzła (przy czym bieżemy pod uwagę również pusty podciąg). Kolejność elementów w ciągu odpowiada porządkowi inorder.

Jak łatwo zauważyć, funkcja `MaksymalnyPrzedzial` sprowadza się do odczytania wartości pola `MiotlaMaxSuma` dla korzenia. Wstawianie wartości do miotły jest trochę bardziej kłopotliwe, należy bowiem aktualizować wszystkie te pola.
Dla danego klucza x i wartości w którą chcemy dodać, należy:

- znaleźć węzeł v zawierający klucz x,
- dodać do pola `MiotlaSuma` wartość w, dla wszystkich węzłów na ścieżce z v do korzenia,
- poprawić wartości `MiotlaMaxSuma` na tej samej ścieżce.

Przy aktualizowaniu pola `MiotlaMaxSuma` w węźle v mogą zajść dwa przypadki:

- podciąg o maksymalnej sumie kończy się w pewnym węźle v', o mniejszym kluczu niż ten z v (a więc znajdującym się w lewym poddrzewie) — w takim wypadku suma elementów takiego podciągu zapisana jest w polu `MiotlaMaxSuma` lewego syna węzła v;
- podciąg o maksymalnej sumie kończy się w pewnym węźle v', o większym lub równym kluczu niż ten z v — w takim wypadku, aby wyznaczyć wartość sumy elementów takiego podciągu należy dodać sumę elementów lewego poddrzewa do wartości przypisanych do węzła v (a więc wartość `MiotlaSuma` z węzła v po odjęciu sum z lewego i prawego poddrzewa) i dodać maksymalną sumę prefiksową z prawego poddrzewa.

Aktualizacja polega na wyliczeniu wartości obu podciągów i zapisaniu w węźle v wartości większego z nich.
Kod procedury realizującej dodawanie wartości do węzła drzewa wygląda następująco:

```
procedure WstawDoMiotly( x, wartosc: Integer);
begin
  { szukanie węzła z kluczem x }
   Wezel := 1;
  repeat
    if x < MiotlaX[ Wezel] then
        Wezel := Wezel * 2 { idziemy do lewego poddrzewa }
    else if x > MiotlaX[ Wezel] then
        Wezel := Wezel * 2 + 1 { idziemy do prawego poddrzewa }
    else
      break;
  until false;
  { Uaktualnianie sum na scieżce od Wezel do korzenia }
  repeat
    if Wezel * 2+1 ≤ IlePktowMiotly then begin
```

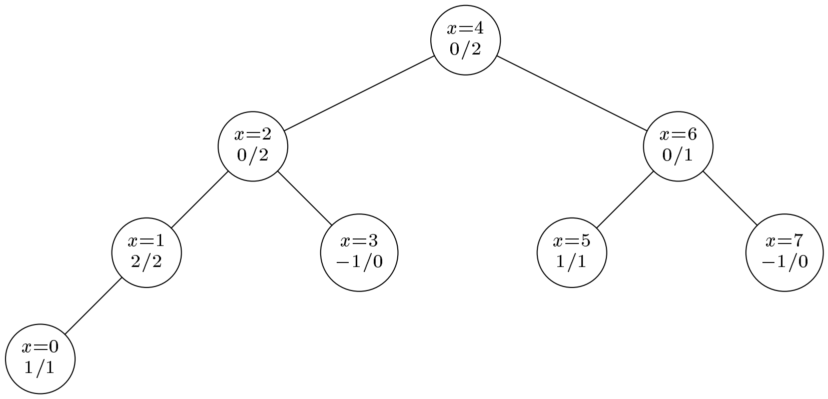
```
      MaxSumaPrawa := MiotlaMaxSuma[ Wezel * 2+1];
      SumaPrawa := MiotlaSuma[ Wezel * 2+1];
    end else begin { jeśli brak prawego syna }
      MaxSumaPrawa := 0; SumaPrawa := 0;
    end;
    if Wezel * 2 ≤ IlePktowMiotly then
      MaxSumaLewa := MiotlaMaxSuma[ Wezel * 2];
    else { jeśli brak lewego syna }
      MaxSumaLewa := 0;
    Inc( MiotlaSuma[ Wezel], k);
    MiotlaMaxSuma[ Wezel] := max(
      MaxSumaLewa, MiotlaSuma[ Wezel]− SumaPrawa+ MaxSumaPrawa);
    Wezel := Wezel div 2 { przejście do ojca w drzewie }
  until Wezel = 0;
end
```

Teraz można wprowadzić zmodyfikowaną procedurę zamiatania:

```
procedure Zamiataj;
begin
  rozw := 0;
  Q := { posortowane trójki wg. wsp. Y i znaku }
  { przetwarzanie samorodków zgodnie z rosnącą wsp. Y }
  while not empty Q do begin
    ( x, y, znak) := Q. extract;
    WstawDoMiotly( x, y, znak);
    WstawDoMiotly( x+szerokosc+1, y, −znak);
    rozw := max( rozw, MiotlaMaxSuma[1]);
  end
end
```

Rysunek 4 przedstawia stan miotły po przetworzeniu 3 pierwszych samorodków.

Rys. 4. Stan drzewa po dodaniu samorodków $(0,0)$, $(5,0)$ i $(1,1)$.



Podsumowując, aby otrzymać pełne rozwiązanie należy połączyć wszystkie wspomniane elementy:

```
PrzygotujMiotle;
PosortujPunkty;
Zamiataj;
```

Do wykonania każdego z tych kroków potrzeba czasu $O(n \log n)$, stąd i całe rozwiązanie ma taką złożoność.

Testy

Rozwiązanie testowane było na 14 zestawach danych testowych, nie stosowano grupowania.

- `kop1.in` — $n = 12$, prosty test poprawnościowy;
- `kop2.in` — $n = 47$, prosty test poprawnościowy;
- `kop3.in` — $n = 48$, prosty test poprawnościowy;
- `kop4.in` — $n = 176$, prosty test poprawnościowy;
- `kop5.in` — $n = 9005$, samorodki ułożone w punktach kratowych (z dodanymi małymi zaburzeniami);
- `kop6.in` — $n = 5000$, samorodki zgrupowane wokół ksztatłtu litery X (z dodanymi małymi zaburzeniami);
- `kop7.in` — $n = 10\,000$, samorodki zgrupowane wokół ksztatłtu litery X (z dodanymi małymi zaburzeniami);
- `kop8.in` — $n = 14\,000$, samorodki zgrupowane wokół ksztatłtu litery X (z dodanymi małymi zaburzeniami);
- `kop9.in` — $n = 15\,000$, samorodki zgrupowane wokół ksztatłtu litery X (z dodanymi małymi zaburzeniami);
- `kop10.in` — $n = 15\,000$, dane losowe;
- `kop11.in` — $n = 15\,000$, dane losowe;
- `kop12.in` — $n = 15\,000$, dane losowe;
- `kop13.in` — $n = 15\,000$, dane losowe;
- `kop14.in` — $n = 15\,000$, dane losowe.

Marcin Kubica
Treść zadania, Opracowanie

Paweł Wolff
Program

Łańcuch

Bajtocja nie zawsze była państwem demokratycznym. W jej historii były również czarne karty. Pewnego razu, generał Bajtelski — przywódca junty żelazną ręką rządzącej Bajtocją — postanowił zakończyć stan wojenny, trwający od momentu przejęcia władzy, i zwolnić więzionych działaczy opozycji. Nie chciał jednak uwolnić przywódcy opozycji Bajtazara. Postanowił przykuć go do murów więzienia za pomocą **bajtockiego łańcucha**. *Bajtocki łańcuch składa się z połączonych ze sobą ogniw oraz przymocowanego do muru pręta. Choć ogniwa nie są połączone z prętem, to bardzo trudno jest je z niego zdjąć.*

– Generale, czemuś mnie przykuł do murów więzienia, miast uwolnić, jako to uczyniłeś z moimi kamratami! — wołał Bajtazar.
– Ależ Bajtazarze, wszak nie jesteś przykuty i z pewnością potrafisz sam zdjąć trzymające Cię ogniwa z pręta wystającego z murów więzienia. — przewrotnie stwierdził generał Bajtelski, po czym dodał — Uporaj się z tym jednak przed godziną cyfracyjną i nie dzwoń łańcuchami po nocy, gdyż w przeciwnym przypadku będę zmuszony wezwać funkcjonariuszy Cyfronicji Obywatelskiej.

Pomóż Bajtazarowi!

Ponumerujmy kolejne ogniwa łańcucha liczbami $1, 2, \ldots, n$. *Ogniwa te możemy zakładać i zdejmować z pręta zgodnie z następującymi zasadami:*

- *jednym ruchem możemy zdjąć lub założyć na pręt tylko jedno ogniwo,*
- *ogniwo nr 1 można zawsze zdjąć lub założyć na pręt,*
- *jeżeli ogniwa o numerach* $1, \ldots, k-1$ *(dla* $1 \le k < n$*) są zdjęte z pręta, a ogniwo nr* k *jest założone, to możemy zdjąć lub założyć ogniwo nr* $k+1$.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `lan.in` *opis bajtockiego łańcucha,*
- *obliczy minimalną liczbę ruchów, które należy wykonać, aby zdjąć wszystkie ogniwa bajtockiego łańcucha z pręta,*
- *zapisze wynik w pliku tekstowym* `lan.out`.

Wejście

W pierwszym wierszu pliku tekstowego `lan.in` *zapisano jedną dodatnią liczbę całkowitą* n, $1 \le n \le 1\,000$. *W drugim wierszu zapisano* n *liczb całkowitych* $o_1, o_2, \ldots, o_n \in \{0, 1\}$ *pooddzielanych pojedynczymi odstępami. Jeśli* $o_i = 1$, *to ogniwo nr* i *jest założone na pręt, a jeśli* $o_i = 0$, *to jest z niego zdjęte.*

Wyjście

Plik tekstowy `lan.out` *powinien zawierać jedną liczbę całkowitą, równą minimalnej liczbie ruchów potrzebnych do zdjęcia wszystkich ogniw bajtockiego łańcucha z pręta.*

Przykład

Dla pliku wejściowego `lan.in`:

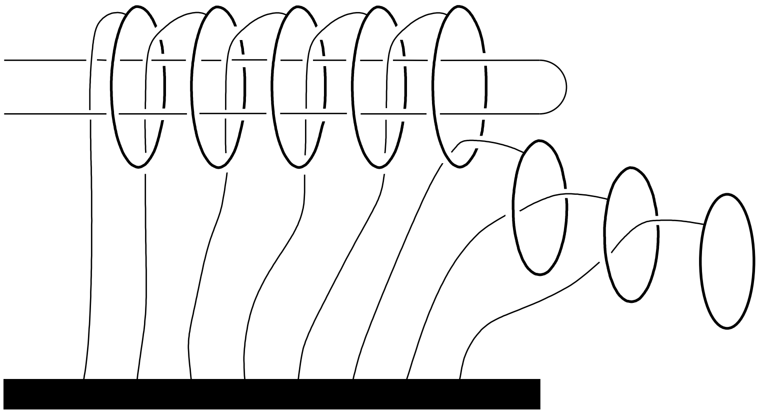
```
4
1 0 1 0
```

poprawną odpowiedzią jest plik wyjściowy `lan.out`:

```
6
```

Analiza problemu

Wielu spośród czytelników zastanawia się zapewne, czy opisany w zadaniu bajtocki łańcuch istnieje w rzeczywistości. Otóż tak, jest to chińska łamigłówka przedstawiona na poniższym rysunku.



Bajtocki łańcuch

Niech $o = (o_1, o_2, \ldots, o_n)$ i $o' = (o'_1, o'_2, \ldots, o'_n)$ będą dowolnymi ciągami (tej samej długości) opisującymi stany bajtockiego łańcucha. Oznaczmy przez $d(o, o')$ minimalną liczbę ruchów potrzebnych do przekształcenia łańcucha opisywanego przez o w łańcuch opisywany przez o'. Zadanie sprowadza się do wyznaczenia $d(o, (0, 0, \ldots, 0))$ dla o będącego ciągiem z pliku wejściowego. Zanim powiemy jak można obliczyć tę wartość, zastanówmy się nad ogólnymi własnościami funkcji d.

Zauważmy, że dla każdego ciągu o mamy $d(o, o) = 0$, gdyż nie trzeba wykonywać żadnych ruchów, aby łańcuch pozostał bez zmian. Oczywiście dla $o \neq o'$ mamy $d(o, o') > 0$. Zauważmy też, że ruchy jakie możemy wykonywać są symetryczne, a więc dla dowolnych o i o' mamy $d(o, o') = d(o', o)$. Ponadto, dla dowolnych o, o' i o'' zachodzi nierówność $d(o, o'') \leq d(o, o') + d(o', o'')$. Jest tak dlatego, że $d(o, o') + d(o', o'')$ jest minimalną liczbą ruchów potrzebnych do przekształcenia łańcucha opisywanego przez o w łańcuch opisywany przez o'', przy czym w trakcie tego przekształcenia, w pewnym momencie łańcuch jest opisywany przez o'. Wszystkie te własności sprawiają, że możemy traktować d jak *odległość* między różnymi *konfiguracjami* łańcucha, przy czym ostatnia z wymienionych własności odpowiada nierówności trójkąta.

Odległość d jest określona dla ciągów dowolnej długości. Zdjęcie lub założenie k-tego ogniwa może wymagać wkładania lub zdejmowania jedynie ogniw $1, 2, \ldots, k$. Położenie ogniw o numerach większych od k nie ma znaczenia dla liczby ruchów potrzebnych do zdjęcia lub założenia k-tego ogniwa. Tak więc, dla dowolnych ciągów o, o' oraz r zachodzi $d(o, o') = d(o \cdot r, o' \cdot r)$[1] Zastanówmy się ile wynosi $d(0^n, 0^{n-1} \cdot 1)$, czyli ile ruchów należy wykonać, aby zdjąć n-te ogniwo, zakładając, że ogniwa 1, 2, ..., $n-1$ są zdjęte i takie mają pozostać. Otóż, żeby zdjąć n-te ogniwo, należy najpierw założyć ogniwo $n-1$ i zdjąć ogniwa 1, 2, ..., $n-2$. Następnie po zdjęciu n-tego ogniwa należy zdjąć również ogniwo $n-1$. Tak więc:

$$\begin{gathered} d(0^n, 0^{n-1} \cdot 1) = \\ d(0^n, 0^{n-2} \cdot 1 \cdot 0) + d(0^{n-2} \cdot 1 \cdot 0, 0^{n-2} \cdot 1 \cdot 1) + d(0^{n-2} \cdot 1 \cdot 1, 0^{n-1} \cdot 1) = \\ d(0^n, 0^{n-2} \cdot 1 \cdot 0) + 1 + d(0^{n-2} \cdot 1 \cdot 1, 0^{n-1} \cdot 1) = \\ d(0^{n-1}, 0^{n-2} \cdot 1) + 1 + d(0^{n-2} \cdot 1, 0^{n-1}) = \\ 2\, d(0^{n-1}, 0^{n-2} \cdot 1) + 1 \end{gathered}$$

Uzyskujemy więc równanie rekurencyjne:

$$d(0^n, 0^{n-1} \cdot 1) = \begin{cases} 2\, d(0^{n-1}, 0^{n-2} \cdot 1) + 1 & \text{dla } n > 1 \\ 1 & \text{dla } n = 1 \end{cases}$$

Jak łatwo sprawdzić, rozwiązanie tego równania to:

$$d(0^n, 0^{n-1} \cdot 1) = 2^n - 1$$

Oznaczmy przez $R(o) = d(o, 0^n)$ oraz $S(o) = d(o, 0^{n-1} \cdot 1)$, gdzie n jest długością ciągu o. Naszym celem jest wyznaczenie wzoru na $R(o)$. Zauważmy, że zachodzą następujące zależności:

$$\begin{gathered} R(o \cdot 0) = R(o) \\ S(o \cdot 1) = R(o) \end{gathered}$$

[1] Symbol $\cdot$ oznacza operację sklejania ciągów. Dla uproszczenia, przez a^n będziemy oznaczać ciąg złożony z n elementów a, $a^n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{n \text{ razy}}$.

Jeżeli ostatnie ogniwo jest założone, to żeby zdjąć wszystkie ogniwa musimy doprowadzić do konfiguracji, w której tylko ostatnie dwa ogniwa są założone, zdjąć ostatnie z nich, a następnie zdjąć przedostatnie:

$$\begin{aligned} R(o \cdot 1) = d(o \cdot 1, 0^{n-1} \cdot 1 \cdot 1) + 1 + d(0^{n-1} \cdot 1 \cdot 0, 0^{n+1}) = \\ d(o, 0^{n-1} \cdot 1) + 1 + d(0^{n-1} \cdot 1, 0^n) = \\ S(o) + 2^n \end{aligned}$$

Podobnie, jeżeli ostatnie ogniwo jest zdjęte, a chcemy doprowadzić do konfiguracji, w której jedynie ostatnie ogniwo jest założone, to musimy doprowadzić do konfiguracji, w której jedynie przedostatnie ogniwo jest założone, założyć ostatnie ogniwo i zdjąć przedostatnie:

$$\begin{aligned} S(o \cdot 0) = d(o \cdot 0, 0^{n-1} \cdot 1 \cdot 0) + 1 + d(0^{n-1} \cdot 1 \cdot 1, 0^n \cdot 1) = \\ d(o, 0^{n-1} \cdot 1) + 1 + d(0^{n-1} \cdot 1, 0^n) = \\ S(o) + 2^n \end{aligned}$$

Uzyskane zależności rekurencyjne mogą być już podstawą rozwiązania zadania. Możemy je jednak jeszcze uprościć. Zauważmy, że $R(o) = S((o_1, o_2, \ldots, 1 - o_n))$. Tak więc:

$$R(o) = \begin{cases} R((o_1, o_2, \ldots, o_{n-1})) & \text{dla } o_n = 0 \\ R((o_1, o_2, \ldots, 1 - o_{n-1})) + 2^{n-1} & \text{dla } o_n = 1 \end{cases}$$

Oznaczmy przez $p_i = (\sum_{j=i}^{n} o_j) \bmod 2$, czyli $p_i = 0$, gdy wśród $o_i, o_{i+1}, \ldots, o_n$ jest parzysta liczba jedynek, w przeciwnym przypadku $p_i = 1$. Wówczas mamy:

$$R(o) = \sum_{i=1}^{n} p_i 2^{i-1}$$

Wygodniej jednak jest zliczać jedynki od początku ciągu, a nie od jego końca. Oznaczmy więc przez $p'_i = \sum_{j=1}^{i} o_j$. Zauważmy, że $p_i = (p'_n - p'_{i-1}) \bmod 2$. Możemy więc wyrazić $R(o)$ następującym wzorem:

$$R(o) = \begin{cases} \sum_{i=1}^{n} p'_{i-1} 2^{i-1} & \texttt{dla } p'_n = 0 \\ \sum_{i=1}^{n} (1 - p'_{i-1}) 2^{i-1} = 2^n - 1 - \sum_{i=1}^{n} p'_{i-1} 2^{i-1} & \texttt{dla } p'_n = 1 \end{cases}$$

Ten wzór będzie podstawą naszego rozwiązania.

Rozwiązanie

Jak widać z analizy problemu, wynik może być bardzo dużą liczbą — może mieć nawet ponad 300 cyfr. Wymaga to zaimplementowania własnej arytmetyki. Przyjmujemy, że zostały zaimplementowane następujące operacje na „dużych liczbach”:

- *przypisz(d, l)* — nadaje dużej liczbie d wartość liczby całkowitej l,
- *dodajjeden(d)* — zwiększa dużą liczbę d o 1,
- *dodaj(d_1, d_2)* — do d_1 dodaje d_2,
- *odejmij(d_1, d_2)* — od d_1 odejmuje d_2,
- *zapiszwynik(p, d)* — zapisuje do pliku p liczbę d.

Wszystkie te operacje można zaimplementować w koszcie (czasowym i pamięciowym) $O(n)$, gdzie n jest liczbą cyfr.

W naszym rozwiązaniu będziemy używać następujących zmiennych. Pliki wejściowy i wyjściowy to odpowiednio *we* i *wy*. Zmienna n to liczba ogniw, o reprezentuje kolejne ogniwa. Zmienna logiczna *dodawac* reprezentuje kolejne wartości p'. Obliczany szereg wyliczamy na zmiennej *wynik*, równocześnie na zmiennej *potega2* obliczamy kolejne potęgi dwójki. Używamy też pomocniczej zmiennej całkowitej i. Oto rozwiązanie:

```
{ Inicjacja zmiennych. }
readln( we, n);
dodawac := false;
przypisz( wynik, 0);
przypisz( potega2, 1);
```

```
{ Sumowanie kolejnych elementów szeregu $\sum_{i=1}^{n} p'_{i-1} 2^{i-1}$. }
for i := 1 to n do begin
  if dodawac then
     dodaj( wynik, potega2);
   dodaj( potega2, potega2);
  read( we, o);
  if o = 1 then
     dodawac := not dodawac
end;
{ Obliczenie i zapisanie wyniku. }
if dodawac then begin
   dodajjeden( wynik);
   odejmij( potega2, wynik);
   zapiszwynik( wy, potega2)
end else
   zapiszwynik( wy, wynik);
```

Zauważmy, że jeśli n jest długością danego ciągu, to liczba cyfr w wyniku jest rzędu $O(n)$, a liczba cyfr w największej obliczonej potędze 2 jest rzędu $\Theta(n)$. Tak więc złożoność pamięciowa programu jest rzędu $\Theta(n)$. Program ten wykonuje $\Theta(n)$ operacji, w tym operacje arytmetyczne na dużych liczbach. W rezultacie złożoność czasowa programu jest rzędu $\Theta(n^2)$.

Pełne rozwiązanie można znaleźć na załączonej dyskietce w pliku `lan.pas`.

Testy

Rozwiązania zawodników sprawdzano na 12 testach. Cztery z tych testów badały przypadki brzegowe: test nr 1 to łańcuch złożony z jednego założonego ogniwa, test nr 2 to trzy zdjęte ogniwa, test nr 8 to 100 ogniw, wszystkie założone, test nr 12 to 1 000 ogniw i tylko ostatnie z nich jest założone. Pozostałe testy to różne testy losowe. Wielkości testów przedstawiono w poniższej tabelce.

Nr testu	n	Wynik
1	1	1
2	3	0
3	9	410
4	25	7 568 677
5	28	167 445 844
6	30	390 843 256
7	100	1 111 099 096 870 146 813 528 342 860 237
8	100	845 100 400 152 152 934 331 135 470 250
9	300	liczba 89-cyfrowa
10	500	liczba 151-cyfrowa
11	1 000	liczba 301-cyfrowa
12	1 000	liczba 302-cyfrowa