Michał Włodarczyk
Treść zadania, Opracowanie

Bartosz Kostka
Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień próbny, 12.04.2016*

Równoważne programy

Bajtazar dostał nowy komputer i uczy się go programować. Program składa się z ciągu instrukcji. Jest k różnych rodzajów instrukcji, które dla uproszczenia oznaczamy liczbami od 1 do k. Niektóre pary instrukcji mają tę własność, że jeśli występują w programie bezpośrednio obok siebie (w dowolnej kolejności), to zamieniając je miejscami, nie zmienia się działania programu (czyli uzyskuje się program **równoważny***). Pozostałe pary instrukcji nie mają tej własności i nazywamy je parami* **nieprzemiennymi***. Bajtazar napisał dwa programy o długości n instrukcji każdy i zastanawia się, czy są one równoważne. Pomóż mu!*

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite n, k oraz m pooddzielane pojedynczymi odstępami, oznaczające odpowiednio długość programów, liczbę różnych instrukcji komputera oraz liczbę par instrukcji nieprzemiennych.

Kolejne m wierszy zawiera opis tych par: każdy z tych wierszy zawiera dwie liczby całkowite a i b ($1 \leqslant a < b \leqslant k$) oddzielone pojedynczym odstępem, oznaczające, że para instrukcji o numerach a i b jest nieprzemienna. Możesz założyć, że każda para wystąpi w tym opisie co najwyżej raz.

Kolejne dwa wiersze przedstawiają opisy dwóch programów. Każdy z tych wierszy zawiera ciąg n liczb całkowitych $c_1, c_2, \ldots, c_n$ ($1 \leqslant c_i \leqslant k$) pooddzielanych pojedynczymi odstępami, oznaczających numery kolejnych instrukcji programu.

Wyjście

W jedynym wierszu standardowego wyjścia należy wypisać jedno słowo `TAK` *lub* `NIE` *w zależności od tego, czy podane na wejściu programy są równoważne.*

Przykład

Dla danych wejściowych:

```
5 3 1
2 3
1 1 2 1 3
1 2 3 1 1
```

poprawnym wynikiem jest:

```
TAK
```

natomiast dla danych wejściowych:

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

poprawnym wynikiem jest:

```
NIE
```

Wyjaśnienie do pierwszego przykładu: *W pierwszym programie można zamienić instrukcje na pozycjach 2 i 3, a następnie instrukcję na pozycji 5 z instrukcjami na pozycjach 4 i 3. W ten sposób uzyska się drugi program.*

Testy „ocen":

`1ocen`: $n = 50$, $k = 50$, $m = 1$; *programy to* $(1, 2, \ldots, 49, 50)$ *oraz* $(50, 49, \ldots, 2, 1)$; *odpowiedź* `NIE`.

`2ocen`: $n = 99\,999$, $k = 3$, $m = 1$; *instrukcje nieprzemienne to 1 i 2, a programy to* $(1, 2, 3, 1, 2, 3, \ldots, 1, 2, 3)$ *oraz* $(3, 1, 2, 3, 1, 2, \ldots, 3, 1, 2)$; *odpowiedź* `TAK`.

`3ocen`: $n = 100\,000$, $k = 1000$, $m = 50\,000$; *programy to* $(13, 13, 13, \ldots, 13)$ *oraz* $(37, 37, 37, \ldots, 37)$; *odpowiedź to oczywiście* `NIE`.

Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów. We wszystkich testach zachodzą warunki $1 \leqslant n \leqslant 100\,000$, $1 \leqslant k \leqslant 1000$, $0 \leqslant m \leqslant 50\,000$.

Podzadanie	Warunki	Liczba punktów
1	$n \leqslant 5$	*5*
2	$k \leqslant 2$	*5*
3	$n \leqslant 1000$	*25*
4	*brak dodatkowych warunków*	*65*

Rozwiązanie

Zadanie polega na rozstrzygnięciu, czy z jednego z danych programów da się otrzymać drugi za pomocą ciągu zamian sąsiednich instrukcji. Przeszkodą jest lista par instrukcji, których nie wolno ze sobą zamieniać. Jeśli istnieje ciąg zamian przekształcający jeden program w drugi, to programy nazywamy *równoważnymi*. Taka nazwa została wybrana nieprzypadkowo. Pojęcie *relacji równoważności* występuje powszechnie w matematyce i jest uogólnieniem relacji opisanej w zadaniu. Wprawdzie znajomość definicji relacji równoważności nie pomaga w żaden szczególny sposób w wymyśleniu efektywnego algorytmu, ale dostarcza języka pomocnego przy opisie rozwiązań.

Definicja 1. Relację $\approx$ nazywamy relacją równoważności, jeżeli jest ona:

1. **zwrotna**, czyli $x \approx x$ dla każdego x,
2. **symetryczna**, czyli $x \approx y$ zachodzi wtedy i tylko wtedy, gdy $y \approx x$,
3. **przechodnia**, czyli $x \approx y$ w połączeniu z $y \approx z$ implikuje $x \approx z$.

Przykładami relacji równoważności są: równość (znak $\approx$ zastępujemy wtedy znakiem $=$), posiadanie takiej samej reszty z dzielenia przez ustalony dzielnik (relacja równoważności określona na liczbach naturalnych) albo podobieństwo figur na płaszczyźnie. Zauważmy też, że opisana w treści zadania równoważność jest relacją równoważności na zbiorze programów. Jeśli oznaczymy ją przez $\approx$, a pod x, y, z podstawimy dowolne programy, to wszystkie trzy warunki powyższej definicji będą spełnione.

Przykładem relacji zwrotnej i symetrycznej, lecz niekoniecznie przechodniej, jest relacja przemienności instrukcji z zadania. Istotnie, może się okazać, że instrukcja p jest w relacji z instrukcjami q i r i może być zamieniona miejscami z każdą z nich, ale kolejność instrukcji q i r ma znaczenie. Za przykład relacji, która nie jest zwrotna ani symetryczna, lecz jest przechodnia, może posłużyć relacja mniejszości liczb ($<$).

Sortowanie z przeszkodami

W rozwiązaniu skorzystamy z przechodniości relacji równoważności i przekształcimy oba programy do prostszych równoważnych postaci, które będziemy umieli łatwo porównać. Gdyby wszystkie pary instrukcji były przemienne, wystarczyłoby posortować numery instrukcji w każdym programie[1] i sprawdzić, czy otrzymaliśmy takie same ciągi. Okazuje się, że podobny pomysł może zadziałać w ogólniejszym przypadku. Niech x' oznacza najmniejszy w porządku leksykograficznym program równoważny z x. Analogicznie dla programu y definiujemy y'. Jeśli $x \approx y$ (programy x i y są równoważne), to z przechodniości relacji równoważności mamy $x' \approx y'$, co z definicji daje $x' = y'$. Z drugiej strony, jeżeli $x' = y'$, to z przechodniości relacji wnioskujemy, że $x \approx y$.

Przykład 1. Zakładając, że nieprzemienne są pary instrukcji 1, 2 oraz 1, 3, następujące programy są równoważne:

$$x = 1,\ 4,\ 2,\ 3,\ 2,\ 1,\ 4,\ 2,\ 1,\ 4 \quad \text{oraz} \quad y = 4,\ 1,\ 4,\ 3,\ 2,\ 2,\ 1,\ 2,\ 1,\ 4.$$

Najmniejszy leksykograficznie program równoważny x oraz y to:

$$x' = y' = 1,\ 2,\ 2,\ 3,\ 1,\ 2,\ 1,\ 4,\ 4,\ 4.$$

Zanim zaczniemy konstruować algorytm, zauważmy, że mając dane dwa numery instrukcji, umiemy w czasie stałym rozstrzygnąć, czy odpowiadające im instrukcje są przemienne. Możemy chociażby trzymać wszystkie nieprzemienne pary w tablicy haszującej. Nie musimy jednak posuwać się do takich optymalizacji – jako że numery instrukcji należą do przedziału $[1, k]$ gdzie $k \leqslant 1000$, bez problemu zmieścimy w pamięci zwykłą tablicę dwuwymiarową indeksowaną numerami instrukcji.

Jak znaleźć najmniejszy leksykograficznie program równoważny z x? Najprościej zacząć od sprawdzenia, czy na pierwszą pozycję można wstawić instrukcję o numerze 1. Aby było to możliwe, taka instrukcja musi występować w programie i wszystkie instrukcje przed nią muszą być z nią przemienne. Jeśli nie jest to możliwe, sprawdzamy instrukcję o numerze 2 i tak dalej. Gdy znajdziemy pierwszą pasującą instrukcję,

[1] Wykorzystując algorytm sortowania przez zliczanie, można to wykonać w czasie $O(n + k)$.

usuwamy ją z programu i powtarzamy proces dla pozycji 2. Dla każdej z n pozycji musimy sprawdzić potencjalnie k kandydatów. Sprawdzenie jednego kandydata wymaga czasu $O(n)$. Prowadzi to do złożoności obliczeniowej algorytmu $O(n^2k)$, co jest dalece niesatysfakcjonujące.

Aby usprawnić algorytm, zawęźmy zbiór kandydatów na pierwszą instrukcję programu. Mogą to być jedynie te instrukcje, których nie poprzedzają żadne nieprzemienne z nimi. Możemy wyznaczyć taki zbiór w czasie $O(n^2)$. Po wyborze najmniejszej wartości na pierwszą pozycję, chcielibyśmy szybko uaktualnić zbiór kandydatów w celu wyłonienia najlepszej instrukcji na pozycję 2 i kontynuować ten proces, aż skonstruujemy x'.

Niech $S^x[i]$ oznacza liczbę instrukcji leżących przed pozycją i w programie x nieprzemiennych z instrukcją $x[i]$. Początkowo kandydaci znajdują się na pozycjach spełniających $S^x[i] = 0$. Kiedy wybierzemy instrukcję na początek programu (niech pochodzi ona z pozycji i_0), przestaje ona blokować dokładnie te instrukcje na pozycjach $i > i_0$, które nie są z nią przemienne, więc możemy uaktualnić dla nich wartości $S^x[i]$. Tym razem pozycje o $S^x[i]$ równym 0 będą zawierać kandydatów do przesunięcia na pozycję 2 i tak dalej.

Algorytm można najszybciej zrozumieć przez analizę poniższego pseudokodu. Gotowy kod znajduje się w pliku `rows3.cpp`.

```
1:  begin
2:    for i := 1 to n do begin
3:      for j := 1 to i − 1 do
4:        if not commute(x[i], x[j]) then
5:          S^x[i] := S^x[i] + 1;
6:      if S^x[i] = 0 then
7:        candidates_x.insert(i);
8:    end
9:    for pos := 1 to n do begin
10:     i_0 := candidates_x.getMin();
11:     candidates_x.popMin();
12:     x'[pos] := x[i_0];
13:     for i := i_0 + 1 to n do
14:       if not commute(x[i_0], x[i]) then begin
15:         S^x[i] := S^x[i] − 1;
16:         if S^x[i] = 0 then
17:           candidates_x.insert(i);
18:       end
19:   end
20: end
```

W pseudokodzie zakładamy, że tablica S^x jest początkowo wyzerowana oraz że dysponujemy funkcją *commute* rozstrzygającą, czy dane instrukcje są przemienne. Ponadto korzystamy ze struktury danych $candidates_x$ obsługującej następujące operacje:

- *insert*(i) : dodaje element i do struktury,
- *getMin*() : zwraca element i o najmniejszej wartości $x[i]$,

- *popMin*() : usuwa element zwracany przez *getMin*().

Najprostszą strukturą danych implementującą powyższe operacje jest lista. Możemy dodać do niej nowy element w czasie stałym, a wyszukiwanie najlepszego kandydata zajmuje czas liniowy. Zazwyczaj na zawodach algorytmicznych lepiej sprawdza się *kolejka priorytetowa*, pozwalająca wykonać wszystkie operacje w czasie co najwyżej logarytmicznym. Zauważmy jednak, że liczba wywołań funkcji *commute* jest rzędu $\Theta(n^2)$, a operacje na strukturze danych wywoływane są jedynie $O(n)$ razy (każdy element zostaje dodany co najwyżej raz). Zatem niezależnie od wyboru struktury danych otrzymujemy rozwiązanie działające w złożoności obliczeniowej $\Theta(n^2)$.

Przedstawiony algorytm można zoptymalizować tak, aby wykonywał jedynie $O(nk)$ operacji, przy wykorzystaniu faktu, że dla każdego rodzaju instrukcji warto rozważać jedynie jej najwcześniejsze wystąpienie w programie. Takie rozwiązanie wymaga jednak większej staranności w doborze struktur danych; jego opis pomijamy. Zamiast tego w następnej sekcji przedstawiamy odmienne rozwiązanie o takiej samej złożoności obliczeniowej, które posiada elegancki dowód poprawności i jest proste w implementacji. Niemniej jednak zachęcamy Czytelnika do próby wymyślenia szybszego sposobu obliczania x'.

Potęga niezmienników

Rozwiązanie wzorcowe wykorzystuje ciekawą własność relacji równoważności, jaką jest występowanie *niezmienników*.

Definicja 2. Dla relacji $\approx$ określonej na zbiorze X niezmiennikiem nazywamy funkcję $f : X \to Y$, spełniającą warunek $x \approx y \Rightarrow f(x) = f(y)$. Jeśli implikacja zachodzi w obie strony, to niezmiennik nazwiemy *silnym*.

Przykładami niezmienników są liczba kątów dla relacji podobieństwa wielokątów albo naiwnie posortowany ciąg instrukcji dla relacji z zadania. Łatwo znaleźć przykłady na to, że żaden z nich nie jest silny. Silnymi niezmiennikami są za to reszta z dzielenia przez p dla relacji przystawania modulo p tudzież uporządkowany ciąg kątów wewnętrznych dla relacji podobieństwa wielokątów. Analizowane w poprzedniej sekcji przyporządkowanie najmniejszego leksykograficznie równoważnego programu z definicji stanowi silny niezmiennik. Niezmiennik, którego chcemy użyć w rozwiązaniu wzorcowym, wymaga bardziej zaawansowanej konstrukcji.

Definicja 3. Dla programu x oraz numeru instrukcji p konstruujemy ciąg x_p następująco:

(1) zastępujemy każde wystąpienie p literą X,

(2) dopisujemy X na początku i na końcu programu x,

(3) pomiędzy każdy parą kolejnych liter X liczymy instrukcje nieprzemienne z p,

(4) tworzymy ciąg x_p, zapisując kolejno wartości obliczone w punkcie (3).

Twierdzenie 1. *Rodzina ciągów $(x_p)_{p=1}^k$ stanowi silny niezmiennik równoważności programów. Innymi słowy, programy x, y są równoważne wtedy i tylko wtedy, gdy dla każdego p zachodzi $x_p = y_p$.*

W powyższym twierdzeniu dwa ciągi uznajemy za równe, jeśli mają tyle samo elementów i elementy o tych samych indeksach są równe.

Dowód: Implikacja $\Rightarrow$ (wykazanie, że przyporządkowanie jest niezmiennikiem). Jeżeli x i y są równoważne, to istnieje sekwencja zamian sąsiednich instrukcji, które są przemienne, przeprowadzająca pierwszy program na drugi. Wystarczy wobec tego zauważyć, że zamiana przemiennych instrukcji nie może zmodyfikować żadnej wartości w żadnym ciągu x_p.

Implikacja $\Leftarrow$. Indukcja po liczbie instrukcji w programie. Programy x, y o długości 1 są równoważne wtedy i tylko wtedy, gdy składają się z tej samej instrukcji q. Dla tejże instrukcji zachodzi $x_q = y_q = (0, 0)$, dla pozostałych mamy zaś $x_p = y_p = (0)$.

Przypuśćmy teraz, że $n > 1$ oraz dla każdego p zachodzi $x_p = y_p$. Niech q będzie pierwszą instrukcją w programie x. Oznacza to, że pierwsza wartość w ciągu x_q (a zarazem y_q) to 0. W takim razie w programie y wszystkie instrukcje znajdujące się przed pierwszym wystąpieniem instrukcji q są z nią przemienne. Niech y' oznacza program otrzymany z y przez przesunięcie pierwszej instrukcji q na początek programu. Oczywiście $y \approx y'$.

Niech x'', y'' oznaczają programy otrzymane z x, y' przez usunięcie początkowej instrukcji q. Zauważmy, że ciąg x''_q różni się od x_q jedynie brakiem początkowego 0 i jest tożsamy z y''_q. Jeśli $p \neq q$ i p jest nieprzemienne z q, to ciągi x''_p, y''_p można otrzymać przez odjęcie 1 od pierwszego wyrazu w ciągach x_p, y'_p. Jeśli zaś p jest przemienne z q, $x''_p = x_p$ i $y''_p = y'_p$. Wobec tego dla każdego p zachodzi $x''_p = y''_p$ i z założenia indukcyjnego wnioskujemy, że $x'' \approx y''$. Dopisanie tej samej instrukcji na początku programu zachowuje równoważność ciągów, zatem $x \approx y'$. Ostatecznie korzystamy z przechodniości, aby otrzymać $x \approx y$. ■

Przykład 2. Dla programu $x = 1,\ 4,\ 2,\ 3,\ 2,\ 1,\ 4,\ 2,\ 1,\ 4$ z przykładu 2 (nieprzemienne są pary instrukcji 1, 2 oraz 1, 3) niezmiennikiem jest:

$$x_1 = (0, 3, 1, 0), \quad x_2 = (1, 0, 1, 1), \quad x_3 = (1, 2), \quad x_4 = (0, 0, 0, 0).$$

Przykładowo, wyznaczając x_1, zapisujemy następujący ciąg:

$$x = \mathrm{X},\ \mathrm{X},\ \cdot,\ \mathrm{N},\ \mathrm{N},\ \mathrm{N},\ \mathrm{X},\ \cdot,\ \mathrm{N},\ \mathrm{X},\ \cdot,\ \mathrm{X},$$

gdzie $\cdot$ oznacza instrukcję przemienną z 1, a N instrukcję nieprzemienną z 1.

Taki sam niezmiennik ma oczywiście program y z przykładu 2.

Rozwiązanie wzorcowe zaimplementowane w pliku `row.cpp` konstruuje wszystkie ciągi x_p, y_p, po czym je porównuje. Wymaga to przeiterowania po obu programach dla każdej wartości $1 \leqslant p \leqslant k$. Jako że umiemy sprawdzić w czasie stałym, czy dwie instrukcje są nieprzemienne, złożoność obliczeniowa algorytmu wynosi $O(nk)$.