Jacek Tomasiewicz
Treść zadania, Opracowanie

Karol Pokorski
Program

Dostępna pamięć: 128 MB.

OI, etap I, 7.10–4.11.2013

Kurierzy

Bajtazar pracuje w firmie BAJ sprzedającej gry komputerowe. Firma BAJ współpracuje z wieloma firmami kurierskimi, które dostarczają sprzedawane gry klientom firmy BAJ. Bajtazar prowadzi kontrolę tego, jak przebiegała współpraca firmy BAJ z firmami kurierskimi. Ma on listę kolejno wysłanych paczek, wraz z informacją o tym, która firma kurierska dostarczyła którą paczkę. Interesuje go, czy któraś z firm kurierskich nie uzyskała niezasłużonej przewagi nad innymi firmami kurierskimi.

Jeżeli w jakimś przedziale czasu określona firma kurierska dostarczyła więcej niż połowę wysłanych wówczas paczek, to powiemy, że firma ta **dominowała** *w tym czasie. Bajtazar chce stwierdzić, czy w określonych przedziałach czasu jakieś firmy kurierskie dominowały, a jeśli tak, to które to były firmy.*

Pomóż Bajtazarowi! Napisz program, który będzie znajdował dominującą firmę lub stwierdzi, że żadna firma nie dominowała.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n i m ($1 \leqslant n, m \leqslant 500\,000$), oddzielone pojedynczym odstępem i oznaczające liczbę wysłanych przez firmę BAJ przesyłek oraz liczbę przedziałów czasowych, dla których chcemy poznać dominujące firmy. Firmy kurierskie są ponumerowane od 1 do n.

Drugi wiersz wejścia zawiera n liczb całkowitych $p_1, p_2, \ldots, p_n$ ($1 \leqslant p_i \leqslant n$), pooddzielanych pojedynczymi odstępami; p_i oznacza numer firmy kurierskiej, która dostarczyła i-tą (w kolejności chronologicznej) wysłaną paczkę.

Kolejne m wierszy zawiera opisy kolejnych zapytań, po jednym w wierszu. Opis każdego zapytania składa się z dwóch liczb całkowitych a i b ($1 \leqslant a \leqslant b \leqslant n$), oddzielonych pojedynczym odstępem, oznaczających, że szukamy firmy dominującej w okresie między wysłaniem a-tej a b-tej paczki włącznie.

W testach wartych łącznie 65% punktów zachodzi dodatkowy warunek $n, m \leqslant 50\,000$, a w testach wartych 30% punktów zachodzi $n, m \leqslant 5000$.

Wyjście

Standardowe wyjście powinno zawierać m wierszy, w których powinny znaleźć się odpowiedzi na kolejne zapytania, po jednej w wierszu. W każdym wierszu powinna znaleźć się jedna liczba całkowita, równa numerowi firmy, która zdominowała rynek w rozważanym przedziale czasu, lub 0, jeśli takiej firmy nie było.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`7 5`	`1`
`1 1 3 2 3 4 3`	`0`
`1 3`	`3`
`1 4`	`0`
`3 7`	`4`
`1 7`	
`6 6`	

Rozwiązanie

Mamy dany ciąg składający się z n elementów. Musimy odpowiedzieć na m zapytań o lidera w pewnym fragmencie ciągu, przy czym *liderem* nazywamy element, który występuje w danym fragmencie więcej niż połowę razy.

Wyszukiwanie lidera

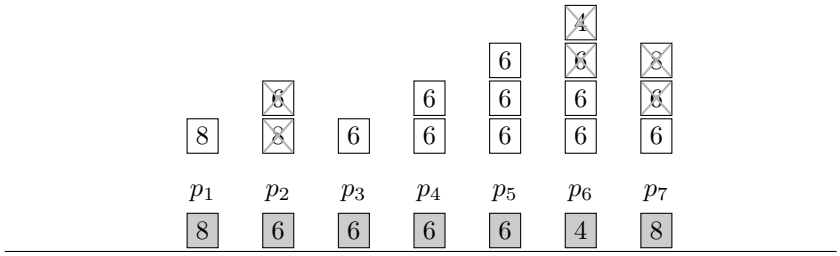
Zauważmy, że jeżeli ciąg długości d ma lidera, to jest on tylko jeden. Gdyby było dwóch liderów, to liczba ich wystąpień musiałaby być większa niż $2 \cdot \frac{d}{2} = d$, a mamy tylko d elementów.

Jeśli dla każdego elementu policzymy jego wystąpienia, przeglądając cały ciąg, to znalezienie lidera zajmie nam czas $O(d^2)$. Jednak lidera możemy znaleźć szybciej. Wystarczy posortować cały ciąg, a następnie zliczać kolejne pakiety elementów o tych samych wartościach. Możemy też to zrobić sprytniej. Zauważmy, że jeśli ciąg ma lidera, to w naszym posortowanym ciągu lider na pewno znajdzie się pod indeksem o numerze $\lceil \frac{d}{2} \rceil$, gdyż nie jesteśmy w stanie zmieścić wszystkich elementów o wartości lidera przed ani za środkowym indeksem. W ten sposób lidera znajdziemy w czasie $O(d \log d)$, ze względu na sortowanie danych.

Spróbujmy uzyskać jeszcze szybsze rozwiązanie. W tym celu przyda nam się pewne proste, ale ważne spostrzeżenie. Zauważmy, że jeżeli ciąg posiada lidera, to usunięcie z niego pary **różnych** elementów da ciąg z tym samym liderem. Faktycznie, ponieważ zawsze usuwamy dwa różne elementy, to najwyżej jeden z nich jest liderem. To oznacza, że lider oryginalnego ciągu w nowym ciągu występuje więcej niż $\frac{d}{2} - 1 = \frac{d-2}{2}$ razy. A zatem jest też liderem nowego ciągu, mającego $d - 2$ elementy.

Usuwanie z ciągu par różnych elementów nie jest zupełnie trywialne. Stworzymy początkowo pusty stos, na który będziemy wkładać kolejne elementy ciągu. Po każdym wstawieniu elementu sprawdzamy, czy na szczycie stosu znajdują się dwie różne wartości. Jeśli tak, to zdejmujemy je ze stosu – jest to równoważne z usunięciem pary różnych liczb z ciągu (patrz rys. 1).

Zauważmy jednak, że tak naprawdę nie musimy trzymać całego stosu, gdyż wszystkie wartości poniżej szczytu będą zawsze takie same. Wystarczy więc przechowywać wartość tych elementów oraz rozmiar stosu (zmienne *wartość* i *krotność*). W ten sposób otrzymujemy następujący pseudokod rozwiązania.



Rys. 1: Przykład działania algorytmu wyznaczania kandydata na lidera za pomocą stosu dla ciągu 8, 6, 6, 6, 6, 4, 8.

```
1:  function kandydatNaLidera(P[1..d])
2:  begin
3:    wartosc := −1;
4:    krotnosc := 0;
5:    for k := 1 to d do
6:      if krotnosc = 0 then begin
7:        krotnosc := 1;
8:        wartosc := P[k];
9:      end else begin
10:       if wartosc ≠ P[k] then
11:         krotnosc := krotnosc − 1;
12:       else
13:         krotnosc := krotnosc + 1;
14:     end
15:   kandydat := −1;
16:   if krotnosc > 0 then
17:     kandydat := wartosc;
18:   return kandydat;
19: end
```

Wcześniej zauważyliśmy, że jeśli początkowy ciąg posiada lidera, to po usunięciu pary różnych elementów lider się nie zmieni. Po usunięciu wszystkich par różnych elementów otrzymamy ciąg stały. Dowolny z elementów tego ciągu na pewno jest kandydatem na lidera, ale nie mamy jeszcze pewności, czy rzeczywiście jest liderem. Aby się o tym przekonać, na koniec powinniśmy przejrzeć cały ciąg i policzyć wystąpienia kandydata – jeśli jest ich więcej niż $\frac{d}{2}$, to znaleźliśmy lidera, a w przeciwnym przypadku ciąg nie posiada lidera.

Złożoność takiego rozwiązania to $O(d)$, ponieważ każdy element rozpatrywany jest tylko raz, a końcowe zliczenie wystąpień kandydata działa również w czasie $O(d)$.

Rozwiązanie wolne $O(n \cdot m)$

Dla każdego zapytania możemy zastosować wprost wyżej opisany algorytm wyszukiwania lidera. W pesymistycznym przypadku będziemy musieli odpowiedzieć na m

pytań o lidera we fragmentach długości $O(n)$, więc uzyskamy złożoność $O(n \cdot m)$. Takie rozwiązanie pozwalało otrzymać około 30% punktów. Przykładowa implementacja znajduje się w plikach `kurs5.cpp` i `kurs6.pas`.

Rozwiązanie wzorcowe $O(n + m \log n)$

Zauważmy, że jeżeli mamy pewne dwa sąsiadujące ze sobą fragmenty ciągu, dla których wyznaczyliśmy wartość i krotność kandydata na lidera, to możemy w łatwy sposób obliczyć analogiczne wyniki dla sumy tych fragmentów. Wartością i krotnością kandydata w scalonym fragmencie będą odpowiednio wartość częściej występującego elementu z fragmentów składowych oraz suma lub różnica krotności, w zależności od tego, czy kandydaci byli równi.

Chcielibyśmy wybrać strukturę danych, która przechowa wartości i krotności dla pewnych *bazowych* fragmentów ciągu (przykładowo, fragmentów o długościach będących potęgami dwójki). Struktura powinna umożliwiać szybkie rozbicie dowolnego fragmentu z zapytania na fragmenty bazowe, dla których są już obliczone wartości i krotności. Dodatkowo, liczba fragmentów bazowych w rozbiciu powinna być nieduża, gdyż scalenie ich wszystkich zajmuje czas liniowy względem ich liczby.

Struktura danych

Idealną strukturą danych do wykonywania żądanych operacji jest **drzewo przedziałowe** (opisane dokładnie w rozwiązaniach zadań *Tetris 3D* z XIII Olimpiady Informatycznej [13] oraz *Koleje* z IX Olimpiady Informatycznej [9], a także w *Wykładach z Algorytmiki Stosowanej*, `http://was.zaa.mimuw.edu.pl`). Tę strukturę danych dla całego ciągu zbudujemy w czasie $O(n)$, a rozbicie dowolnego fragmentu na fragmenty bazowe wykonamy za jej pomocą w czasie $O(\log n)$, uzyskując maksymalnie $O(\log n)$ fragmentów bazowych.

Przetworzenie kandydatów

Drzewo przedziałowe umożliwi szybkie znalezienie kandydata na lidera w każdym fragmencie. W ten sposób uzyskamy m fragmentów z przypisanymi kandydatami na lidera. Chcielibyśmy szybko obliczyć liczbę wystąpień każdego kandydata w jego fragmencie. Jeśli wykonamy to siłowo, uzyskamy złożoność $O(n \cdot m)$.

Wszystkich kandydatów możemy jednak przetworzyć hurtowo. Będziemy przeglądać cały ciąg od lewej do prawej i zliczać wystąpienia poszczególnych liczb. Niewielki zakres wartości elementów pozwala nam utworzyć tablicę, w której każda z liczb może być zliczona pod indeksem odpowiadającym jej wartości.

Jak wykorzystać powyższą tablicę do wyznaczenia liczby wystąpień kandydata w dowolnym fragmencie $[x..y]$? Jeśli napotykamy koniec fragmentu w miejscu y, to w tym momencie pod odpowiednim indeksem w tablicy mamy informację o liczbie wystąpień kandydata we fragmencie $[1..y]$, czyli trochę za dużo. Wcześniej, napotykając początek fragmentu, moglibyśmy zapamiętać liczbę wystąpień kandydata we fragmencie $[1..(x-1)]$. Odejmując liczby wystąpień kandydata w powyższych fragmentach uzyskamy liczbę wystąpień kandydata w szukanym fragmencie $[x..y]$.

Przetworzenie wszystkich kandydatów zajmie czas $O(n + m)$. Ostatecznie, złożoność czasowa całego algorytmu wyniesie $O(n + m \log n)$, przy czym $O(m \log n)$ to czas rozbicia m fragmentów na fragmenty bazowe. Implementacja tego rozwiązania znajduje się w plikach `kur.cpp` i `kur2.pas`.

Rozwiązanie randomizowane $O(k \cdot (n + m))$

Spróbujmy wymyślić inny sposób znajdowania kandydatów na lidera. Dla pojedynczego fragmentu możemy takiego kandydata po prostu wylosować. Zauważmy, że wylosowany element będzie poprawnym kandydatem z prawdopodobieństwem większym od 50%. Rzeczywiście, jeśli fragment zawiera lidera, to lider ten występuje we fragmencie więcej niż połowę razy, natomiast jeśli we fragmencie lidera nie ma, to każdy kandydat jest równie dobry.

Tak więc prawdopodobieństwo tego, że się pomylimy, jest mniejsze niż $\frac{1}{2}$. Możemy je poprawić, jeśli wylosujemy k kandydatów i dla każdego z nich sprawdzimy, czy rzeczywiście jest liderem. Prawdopodobieństwo błędu wynosi wtedy mniej niż $(\frac{1}{2})^k$. Dla ustalonego k, powiedzmy $k = 10$, prawdopodobieństwo błędu dla pojedynczego fragmentu wyniesie maksymalnie $(\frac{1}{2})^{10} = \frac{1}{1024} \approx 0{,}1\%$, czyli stosunkowo mało. Tak więc prawdopodobieństwo sukcesu będzie co najmniej $1 - (\frac{1}{2})^{10} \approx 99{,}9\%$.

Zauważmy, że oszacowaliśmy prawdopodobieństwo dla pojedynczego zapytania, a w zadaniu mamy m zapytań. Jeśli dla każdego fragmentu wylosujemy oddzielnie k kandydatów, to prawdopodobieństwo sukcesu wyniesie co najmniej $(1 - (\frac{1}{2})^k)^m$. Okazuje się, że już dla $k = 26$ prawdopodobieństwo sukcesu dla maksymalnego $m = 500\,000$ to w najgorszym razie około 99%, co możemy uznać za satysfakcjonujące.

W ten sposób zgromadziliśmy po k kandydatów dla każdego fragmentu. Musimy teraz sprawdzić, czy któryś z nich jest faktycznie liderem. Wszystkich kandydatów możemy przetworzyć za pomocą sposobu opisanego w rozwiązaniu wzorcowym: wybieramy po jednym kandydacie z każdego fragmentu i wykonujemy algorytm przetwarzania. Następnie całość powtarzamy jeszcze $k - 1$ razy.

Złożoność czasowa otrzymanego rozwiązania to $O(k \cdot (n + m))$, ze względu na k-krotne przetwarzanie wszystkich fragmentów. Takie rozwiązanie dla odpowiednio dobranego k także uzyskiwało maksymalną liczbę punktów. Implementacja znajduje się w plikach `kur3.cpp` i `kur4.pas`.

Testy

Przygotowano 15 grup testów. Pierwsza grupa składa się z małych testów poprawnościowych, wygenerowanych ręcznie. Wszystkie pozostałe grupy zawierają testy następujących typów:

- naprzemienne ciągi złożone tylko z dwóch wartości z drobnymi losowymi zmianami;
- ciąg posortowany złożony z liczb od 1 do $\sqrt{n}$ z drobnymi zmianami;
- ciągi generowane losowo.