Szymon Acedański **Anna Niewiarowska**
Treść zadania, Opracowanie Program

OI, Etap II, dzień drugi, 08.02.2007

Megalopolis

Globalizacja nie ominęła Bajtocji. Nie ominęła również listonosza Bajtazara, niegdyś chodzącego polnymi drogami pomiędzy wioskami, a dziś pędzącego samochodem po autostradach. Jednak to te dawne spacery Bajtazar wspomina dziś z rozrzewnieniem.

Dawniej n bajtockich wiosek, ponumerowanych od 1 do n, było połączonych dwukierunkowymi polnymi drogami, w taki sposób, że z każdej wioski można było dojść do wioski numer 1 (zwanej Bitowicami) na dokładnie jeden sposób; w dodatku droga ta przechodziła jedynie przez wioski o numerach nie większych niż numer wioski początkowej. Ponadto każda polna droga łączyła dwie różne wioski i nie przechodziła przez żadne inne wioski oprócz tych dwóch. Drogi się nie krzyżowały poza wioskami, lecz mogły istnieć tunele bądź wiadukty.

Z biegiem czasu kolejne polne drogi zamieniano na autostrady. Bajtazar dokładnie pamięta, kiedy każda z polnych dróg została zamieniona w autostradę. Dziś w Bajtocji nie można już spotkać ani jednej polnej drogi — wszystkie zostały zastąpione autostradami, które połączyły wioski w Bajtockie Megalopolis.

Bajtazar pamięta swoje wyprawy do wiosek z listami. Za każdym razem wyruszał z Bitowic, idąc z listami do pewnej innej wioski. Teraz prosi Cię, żebyś dla każdej takiej wyprawy (która miała miejsce w określonym momencie i prowadziła z Bitowic do określonej wioski) policzył, przez ile polnych dróg ona prowadziła.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia:*
  - *opis dróg, które łączyły kiedyś bajtockie wioski,*
  - *sekwencję zdarzeń: wypraw Bajtazara i momentów, gdy poszczególne polne drogi były zamieniane w autostrady,*
- *dla każdej wyprawy obliczy, iloma polnymi drogami Bajtazar musiał przejść,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 250\,000$), oznaczająca liczbę wiosek w Bajtocji. W kolejnych $n-1$ wierszach znajdują się opisy dróg. Każdy z nich składa się z dwóch liczb całkowitych a, b ($1 \leqslant a < b \leqslant n$) oddzielonych pojedynczym odstępem. Są to numery wiosek połączonych drogą.

W kolejnym wierszu znajduje się jedna liczba całkowita m ($1 \leqslant m \leqslant 250\,000$), oznaczająca liczbę wypraw odbytych przez Bajtazara. W kolejnych $n+m-1$ liniach znajdują się opisy zdarzeń, w kolejności chronologicznej:

- *Opis postaci* `A` **a b** *(dla $a < b$) oznacza, że w danym momencie polną drogę pomiędzy wioskami a oraz b zamieniono na autostradę.*
- *Opis postaci* `W` **a** *oznacza, że Bajtazar odbył wyprawę z Bitowic do wioski numer a.*

Wyjście

Na standardowe wyjście Twój program powinien wypisać dokładnie m liczb całkowitych, po jednej w wierszu, oznaczających liczbę polnych dróg, które pokonał Bajtazar w kolejnych wyprawach.

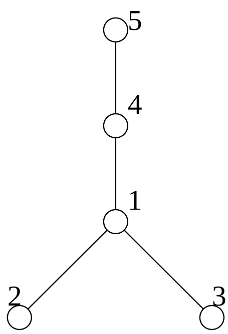
Przykład

Dla danych wejściowych:

```
5
1 2
1 3
1 4
4 5
4
W 5
A 1 4
W 5
A 4 5
W 5
W 2
A 1 2
A 1 3
```

poprawnym wynikiem jest:

```
2
1
0
1
```



Rozwiązanie

Pierwszym krokiem do rozwiązania prawie każdego zadania z Olimpiady Informatycznej jest wypatrzenie problemu algorytmicznego ukrytego za „historyjką". Zapiszmy zatem zadanie *Megalopolis* tak, by problem stał się lepiej widoczny:

Drogi i wioski tworzą drzewo ukorzenione, w którym niektóre krawędzie (autostrady) mogą być wyróżnione. Struktura danych musi umożliwiać wykonywanie dla drzewa następujących operacji:

- *zaznacz(e)* — krawędź e zostaje wyróżniona,
- *ile(v)* — zwraca liczbę niewyróżnionych krawędzi na ścieżce od korzenia do wierzchołka v.

Ewentualnie nieco inaczej:

Drzewo ukorzenione reprezentuje istniejące w danym momencie polne drogi. Struktura danych umożliwia wykonywanie następujących operacji:

- *skróć(e)* — operacja polega na zastąpieniu krawędzi e i jej obu końców przez jeden wierzchołek w grafie, czyli usunięciu z drzewa drogi zamienionej w autostradę;
- *wysokość(v)* — zwraca wysokość wierzchołka v w drzewie, czyli długość ścieżki prowadzącej z wierzchołka v do korzenia.

Proste rozwiązania

Oto kilka prostych sposobów zaimplementowania zaproponowanych struktur danych.

Pierwsza interpretacja. Poniższe procedury są prostą implementacją drzewa z wyróżnionymi krawędziami.

```
function zaznacz(e)
begin
  e.zaznaczona := true;
end

function ile(v)
begin
  wynik := 0;
  while v nie jest korzeniem do
  begin
    if krawędź(v, ojciec(v)) nie jest zaznaczona then
      wynik := wynik + 1;
    v := ojciec(v);
  return wynik;
end
```

Złożoność operacji *zaznacz* jest stała ($O(1)$), ale wyznaczanie wartości *ile* działa w czasie $O(h)$, gdzie h to wysokość drzewa. Niestety, wysokość ta może wynosić nawet $n-1$.

Druga interpretacja. Tworzymy drzewo oddające aktualny układ polnych dróg. Każdy wierzchołek posiada listę swoich dzieci. Operacja *skróć(e)* polega na utożsamieniu obu końców e, co symulujemy, łącząc listy ich dzieci. Do sprawnego operowania na zbiorach dzieci wierzchołka można użyć struktury *Find-Union* (patrz, na przykład, [19]). Operacja *wysokość(v)* może być zaimplementowana przez przejście od v do korzenia i policzenie krawędzi na ścieżce. Wówczas skracanie krawędzi działa w czasie $O(\log^* n)$, a liczenie wysokości (po uwzględnieniu faktu, że niektóre wierzchołki zastępują kilka innych) w czasie $O(h \cdot \log^* n)$.

W obu przedstawionych strukturach jedna z operacji, niestety bardzo często wykonywana w zadaniu, wymaga czasu proporcjonalnego do wysokości drzewa. Chcąc otrzymać efektywne rozwiązanie problemu, musimy to ulepszyć.

Rozwiązanie wzorcowe

Zastanówmy się, czy do rozwiązania problemu nie przydałaby się jakaś ogólna struktura danych — na przykład słownik. W takim słowniku, wraz z wierzchołkami moglibyśmy przechowywać ich wysokość w drzewie. Dodatkowo chcielibyśmy umieć szybko zmniejszyć o jeden wysokości wszystkich wierzchołków w pewnym poddrzewie (to odpowiada operacji *zaznacz()* czy też *skróć()*). Niestety, popularne implementujące słowników (np. drzewa zrównoważone), nie pozwalają na efektywne wykonanie tej operacji. Przydatna okazuje się jednak struktura danych zwana *drzewem licznikowym* (patrz, na przykład, opracowanie zadania *Koleje* w *Niebieskiej Książeczce* z IX OI, [9]). Jest to słownik z tradycyjnymi operacjami *szukaj(klucz)* i *wstaw(klucz, wartość)*, w którym kluczami są liczby ze zbioru $\{1,\ldots,n\}$, dla uprzednio ustalonego n. Dodatkowo drzewo licznikowe posiada operację *zwiększ_w_przedziale(a,b)*, która zwiększa o jeden wszystkie wartości odpowiadające kluczom *z przedziału* $[a,b]$ w czasie $O(\log n)$.

Jeśli udałoby się nam ponumerować wierzchołki w drzewie z treści zadania tak, żeby dla każdego poddrzewa numery wierzchołków tworzyły przedział, to bylibyśmy na drodze prowadzącej wprost do rozwiązania. Zauważmy, że nie jest to trudne:

Obserwacja 1 Jeśli ponumerujemy wierzchołki drzewa w kolejności *preorder*, *inorder* lub *postorder*, to numery wierzchołków wchodzących w skład dowolnego poddrzewa tworzą przedział.

Korzystając z tego spostrzeżenia, możemy zaimplementować operacje wymagane w pierwszej interpretacji w następujący sposób:

```
function inicjuj()
begin
   Ponumeruj wierzchołki w kolejności preorder:
   { Uruchamiamy procedurę DFS i zapisujemy w każdym wierzchołku: }
   { nr — jego numer }
   { h — jego pierwotną wysokość }
   { od,do — przedział numerów wierzchołków w jego poddrzewie };
   Stwórz drzewo licznikowe o n kluczach, we wszystkich wierzchołkach
   (pole val) wpisując wartość zero;
end
```

```
function zaznacz(e)
begin
   zwiększ_w_przedziale(e.od, e.do);
end

function ile(v)
begin
   return v.h − szukaj(v);
end
```

W powyższym rozwiązaniu operacja *inicjuj* działa w czasie $O(n)$, natomiast *zaznacz(e)* oraz *ile(v)* — w czasie $O(\log n)$. Razem daje to złożoność $O((m+n)\log n)$. Rozwiązanie to zostało zapisane w plikach `meg.cpp` i `meg1.pas`.

Rozwiązanie *offline*

Opracowanie rozpoczęliśmy od zinterpretowania zadania w języku teorii grafów. Obie podane interpretacje, choć dość naturalne, narzucają nam podobny sposób widzenia problemu. W obu interpretacjach staramy się na bieżąco symulować operacje wykonywane na sieci dróg i udzielać odpowiedzi na pytania o liczbę polnych dróg zaraz po pojawieniu się zapytania w danych. Tymczasem taki pośpiech nie jest konieczny. Możemy rozpocząć od przeczytania wszystkich danych wejściowych, przeprowadzić obliczenia i na końcu wypisać odpowiedzi na wszystkie pytania. Takie podejście, zwane *rozwiązaniem offline*, może okazać się bardziej eleganckie, prostsze w implementacji, a nawet szybsze (sprawdziło się już wcześniej, na przykład w zadaniu *Małpki* z finału X OI, [10]).

Zacznijmy od wprowadzenia dwóch pojęć: za *moment_zapytania* oraz *moment_zbudowania* przyjmiemy odpowiednio numer wiersza danych wejściowych, w której pojawiło się pytanie lub informacja o budowie autostrady.

W naszym rozwiązaniu offline *drzewo dróg* będziemy reprezentować, zapisując przy każdym wierzchołku listę jego dzieci. Algorytm będzie polegał na przejściu drzewa dróg procedurą DFS. Podczas przechodzenia będziemy utrzymywać następującą strukturę danych:

- *historia budowy*, czyli zbiór zawierający *momenty_zbudowania* wszystkich autostrad na ścieżce od korzenia do aktualnie odwiedzanego wierzchołka — informacje te będą zapisane w zbiorze uporządkowanym, na którym będziemy wykonywać operacje *dodaj(wartość)*, *usuń(wartość)* oraz *ile_wcześniejszych_niż(wartość)*; dobrą implementacją takiej struktury jest zrównoważone drzewo binarne — wszystkie trzy operacje można wykonać w czasie logarytmicznym.

Odwiedzając wierzchołek v i chcąc dowiedzieć się, przez ile autostrad jechał Bajtazar do v w momencie t, wystarczy sprawdzić, ile zdarzeń wcześniejszych niż t jest zapisanych w historii budowy. W ten sposób odpowiedzi na pytania będziemy znajdować w kolejności zgodnej z przechodzeniem drzewa dróg w porządku *preorder* i będziemy je zapisywać w tablicy *odpowiedź* — na i-tej pozycji znajdzie się odpowiedź na pytanie, które pojawiło się w i-tym momencie.

```
function czytaj_dane()
begin
  { Dla każdego wierzchołka znajdujemy: }
  {   jego ojca oraz listę jego dzieci, }
  {   listę momentów_zapytań o ten wierzchołek, }
  {   odległość od korzenia w początkowym drzewie dróg, }
  {   moment_zbudowania autostrady od wierzchołka do jego ojca. }
end

function dfs(v)
begin
  dodaj(moment_zbudowania autostrady (v, ojciec(v)));
  for moment_zapytania in zapytania o v do
    odpowiedź[moment_zapytania] :=
      ile_wcześniejszych_niż(moment_zapytania);
  for u∈dzieci(v) do dfs(u);
  usuń(moment_zbudowania autostrady (v, ojciec(v)));
end

function rozwiąż()
begin
  czytaj_dane();
  dfs(korzeń_drzewa_dróg);
end
```

Po uruchomieniu funkcji *rozwiąż* w tablicy *odpowiedź* otrzymamy szukane wyniki. Czas działania to $O((m+n) \cdot \log h)$, gdzie n oznacza rozmiar drzewa, $h = O(n)$ jest jego wysokością, a m to liczba zapytań.

Zadanie na deser

Wśród rozwiązań zgłoszonych przez zawodników pojawił się ciekawy pomysł. My w pierwotnym rozwiązaniu wzorcowym użyliśmy drzewa licznikowego. Podobne rozwiązanie można jednak uzyskać bez tej struktury — wykorzystując dwa zbiory uporządkowane (drzewa zrównoważone). Zastanów się, jak to zrobić. Może przydać Ci się przy tym poniższa wskazówka:

Obserwacja 2 Przyjmijmy, że wierzchołki drzewa są ponumerowane w kolejności *preorder*. Niech P_i oznacza przedział numerów odpowiadających wierzchołkom w poddrzewie ukorzenionym w wierzchołku i. Wówczas liczba autostrad na drodze z korzenia do wierzchołka v jest równa liczbie takich i, dla których $P_v \subset P_i$ oraz krawędź $(i, \text{ojciec}(i))$ została już przerobiona na autostradę.

Testy

Do oceny przygotowano zestaw czternastu testów, o następujących parametrach:

Nazwa	n	m	Opis
meg1.in	10	15	mały test, krótkie ścieżki w drzewie
meg2.in	30	50	mały test, dłuższe ścieżki w drzewie
meg3.in	50	2 000	mały test, dużo pytań w porównaniu z liczbą krawędzi
meg4.in	100 000	50 000	duży test, bardzo krótkie ścieżki
meg5.in	20 000	30 000	średni test, ścieżki średniej długości
meg6.in	50 000	65 000	średni test, długie ścieżki
meg7.in	150 000	160 001	duży test, ścieżki średniej długości
meg8.in	200 001	170 001	duży test, ścieżki średniej długości
meg9.in	200 000	200 000	duży test, długie ścieżki
meg10.in	200 000	150 000	duży test, długie ścieżki
meg11.in	120 000	250 000	duży test, maksymalna liczba pytań, mniej wiosek
meg12.in	250 000	100 000	duży test, maksymalna liczba wiosek, mniej pytań
meg13.in	250 000	250 000	duży test, maksymalna liczba wiosek i pytań
meg14.in	250 000	250 000	duży test, maksymalna liczba wiosek i pytań