Jacek Tomasiewicz	Bartosz Tarnawski	Piotr Smulewicz
Treść zadania	Opracowanie	Program

Dostępna pamięć: 64 MB. *OI, etap I, 7.10–4.11.2013*

Hotele

W Bajtocji jest n miast połączonych zaledwie $n-1$ drogami. Każda z dróg łączy bezpośrednio dwa miasta. Wszystkie drogi mają taką samą długość i są dwukierunkowe. Wiadomo, że z każdego miasta da się dojechać do każdego innego dokładnie jedną trasą, złożoną z jednej lub większej liczby dróg. Inaczej mówiąc, sieć dróg tworzy **drzewo**.

Król Bajtocji, Bajtazar, chce wybudować trzy luksusowe hotele, które będą gościć turystów z całego świata. Król chciałby, aby hotele znajdowały się w różnych miastach i były położone w tych samych odległościach od siebie.

Pomóż królowi i napisz program, który obliczy, na ile sposobów można wybudować takie trzy hotele w Bajtocji.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 5000$), oznaczającą liczbę miast w Bajtocji. Miasta są ponumerowane od 1 do n.

Sieć dróg Bajtocji jest opisana w kolejnych $n-1$ wierszach. Każdy z tych wierszy zawiera dwie liczby całkowite a i b ($1 \leqslant a < b \leqslant n$) oddzielone pojedynczym odstępem, oznaczające, że miasta a i b są połączone bezpośrednio drogą.

W testach wartych łącznie 50% punktów zachodzi dodatkowy warunek $n \leqslant 500$.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą równą liczbie sposobów wybudowania hoteli.

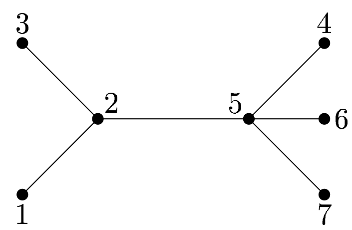
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
5
```



Wyjaśnienie do przykładu: *Trójki (nieuporządkowane) miast, w których można wybudować hotele, to: $\{1,3,5\}$, $\{2,4,6\}$, $\{2,4,7\}$, $\{2,6,7\}$, $\{4,6,7\}$.*

Rozwiązanie

Mamy dane drzewo, nazwijmy je T. Niech $d(u,v)$ oznacza odległość pomiędzy wierzchołkami u i v w tym drzewie (liczbę krawędzi między nimi). Przez $u \rightsquigarrow v$ oznaczymy ścieżkę od u do v. Powiemy, że trzy różne wierzchołki x, y, z są *równoodległe*, jeśli $d(x,y) = d(x,z) = d(y,z)$.

Chcemy znaleźć liczbę nieuporządkowanych trójek równoodległych wierzchołków.

Rozwiązanie bardzo siłowe $O(n^4)$

Dla każdej trójki wierzchołków chcemy sprawdzić, czy są równoodległe. W tym celu wykonujemy przeszukiwanie naszego drzewa z każdego z nich (wszerz lub w głąb). To rozwiązanie zostało zaimplementowane w plikach `hots1.cpp`, `hots2.pas`. Na zawodach zdobywało ok. 30 punktów.

Rozwiązanie siłowe $O(n^3)$

Wykonujemy przeszukiwanie z każdego z wierzchołków i tablicujemy odległości między każdą parą wierzchołków. Przeglądamy wszystkie trójki wierzchołków. Dla każdej z nich sprawdzamy w czasie stałym, czy są one równoodległe.

To rozwiązanie, w odróżnieniu od poprzedniego, używa $O(n^2)$ pamięci. Zostało zaimplementowane w plikach `hotb1.cpp`, `hotb3.pas`. Zgodnie z informacją w treści zadania zdobywało na zawodach ok. 50 punktów.

Rozwiązanie wzorcowe $O(n^2)$

Zaczniemy od prostego lematu, który pomoże nam później lepiej scharakteryzować równoodległe trójki.

Lemat 1. Niech x, y, z będą wierzchołkami drzewa T. Ścieżki $x \rightsquigarrow y$, $y \rightsquigarrow z$ i $z \rightsquigarrow x$ mają dokładnie jeden wierzchołek wspólny.

Dowód: Dla danej trójki wierzchołków x, y, z określmy wierzchołek w jako pierwszy wierzchołek ze ścieżki $z \rightsquigarrow x$, który znajduje się także na ścieżce $y \rightsquigarrow x$. Wykażemy, że jest to dokładnie wierzchołek, którego szukamy.

Z definicji wierzchołka w mamy, że na ścieżce $y \rightsquigarrow w \rightsquigarrow z$ żaden wierzchołek nie powtarza się. (Ścieżka ta może być pojedynczym wierzchołkiem, gdy $y = z$). Jest to więc najkrótsza ścieżka z y do z. A zatem wierzchołek w leży na każdej ze ścieżek $y \rightsquigarrow x$, $z \rightsquigarrow x$ i $y \rightsquigarrow z$. Przecięciem dwóch pierwszych z tych ścieżek jest $w \rightsquigarrow x$, a jedynym punktem wspólnym tej ścieżki ze ścieżką $y \rightsquigarrow w \rightsquigarrow z$ jest w. Tak więc jest to jedyny wierzchołek leżący na przecięciu tych wszystkich ścieżek. ∎

Wierzchołek w określony w lemacie 1 nazwiemy *wierzchołkiem spotkania*. W algorytmie wzorcowym będziemy ukorzeniać nasze drzewo po kolei w każdym jego wierzchołku i zliczać te równoodległe trójki, dla których jest on wierzchołkiem spotkania.

Lemat 2. Jeśli x, y, z jest trójką równoodległych wierzchołków drzewa T, to ich wierzchołek spotkania w spełnia $d(x,w) = d(y,w) = d(z,w)$.

Dowód: Oznaczmy $d_1 = d(x,w)$, $d_2 = d(y,w)$, $d_3 = d(z,w)$. Mamy $d(x,y) = d_1 + d_2$, $d(x,z) = d_1 + d_3$ oraz $d(y,z) = d_2 + d_3$. Z przyrównania tych trzech odległości otrzymujemy, że $d_1 = d_2 = d_3$. ■

Ukorzeńmy więc T w jednym z jego wierzchołków r. Od tej chwili odległość dowolnego wierzchołka v od r nazywamy *głębokością* v. Ustawmy poddrzewa synów wierzchołka r w kolejności nierosnących maksymalnych głębokości. Oznaczmy je $S_1, S_2, \ldots, S_m$.

Będziemy korzystać z następującej obserwacji, będącej konsekwencją lematu 2.

Lemat 3. Niech x, y, z będą różnymi wierzchołkami na tej samej głębokości. Wtedy następujące warunki są równoważne:

1. x, y, z należą do poddrzew trzech różnych synów r,
2. x, y, z są równoodległe i r jest ich wierzchołkiem spotkania.

Niech najgłębszy wierzchołek w poddrzewie S_i będzie na głębokości D_i. Mamy zatem $D_1 \geqslant D_2 \geqslant \ldots \geqslant D_m$. Przeszukujemy każde z poddrzew S_i, wyznaczając tablicę $at\_depth_i[1..D_i]$ taką, że $at\_depth_i[l]$ jest liczbą wierzchołków w S_i o głębokości l. Zauważmy, że przeszukanie poddrzew wszystkich synów korzenia zajmie nam łącznie $O(n)$ czasu.

Ustalmy teraz l ($1 \leqslant l \leqslant D_1$). Chcielibyśmy wyznaczyć liczbę trójek wierzchołków na głębokości l, należących do poddrzew różnych synów r (korzystamy z lematu 3).

Dla dwóch wierzchołków u, v napiszemy $u \prec v$, jeśli $u \in S_{i_u}$, $v \in S_{i_v}$ oraz $i_u < i_v$. Będziemy wybierać po kolei $i = 2, 3, \ldots, m-1$ i zliczać takie trójki wierzchołków x, y, z na głębokości l, że $x \prec y \prec z$ oraz $y \in S_i$. W tym celu wystarczy utrzymywać dwie zmienne $before_i[l]$ oraz $after_i[l]$, aby przy rozważaniu danego i zachodził niezmiennik:

$$before_i[l] = \sum_{a<i} at\_depth_a[l], \quad after_i[l] = \sum_{i<b} at\_depth_b[l].$$

Wówczas szukana liczba trójek wierzchołków x, y, z spełniających $d(x,r) = d(y,r) = d(z,r) = l$, $x \prec y \prec z$, $y \in S_i$ jest równa

$$before_i[l] \cdot at\_depth_i[l] \cdot after_i[l]. \tag{1}$$

Sumując iloczyny (1) dla każdego korzenia r, każdego indeksu syna i oraz każdej głębokości $l \leqslant D_i$, otrzymujemy wynik.

Jaka jest złożoność czasowa rozwiązania przy ustalonym korzeniu r? Najważniejsze spostrzeżenie jest takie, że iloczyn (1) obliczamy tylko wtedy, gdy w poddrzewie S_i rzeczywiście jest jakiś wierzchołek położony na głębokości l. To oznacza, że liczbę obliczeń iloczynów możemy oszacować z góry przez liczbę wszystkich wierzchołków we wszystkich poddrzewach, czyli po prostu przez n. Łatwo zauważyć, że pozostałe obliczenia, tj. wyznaczanie wartości $before_i[l]$ oraz $after_i[l]$, możemy wykonać w takim samym czasie.

Dla ustalonego r zużywamy $O(n)$ czasu, więc ostateczna złożoność to $O(n^2)$. Warto zauważyć, że wynik jest rzędu $O(n^3)$, więc należy go pamiętać w zmiennej całkowitej 64-bitowej.

Powyższy algorytm pozwalał na zdobycie 100 punktów. Jego implementacje znajdują się w plikach `hot.cpp`, `hot2.pas`.

Szybsze rozwiązanie $O(n \log n)$

To rozwiązanie powstało dosyć późno – niemal rok po I etapie XXI Olimpiady. Pod pewnymi względami jest podobne do rozwiązania $O(n^2)$. Tutaj też będziemy ukorzeniać drzewo (ale tylko raz) i skorzystamy z lematu 1. Pewną nowością będzie wykorzystanie *centroidalnej dekompozycji* drzewa (ang. *centroid decomposition*) – przydatnej techniki, która pozwala na przetwarzanie drzewa metodą dziel i zwyciężaj.

Przygotowania

Ponownie ukorzeniamy nasze drzewo w r – jednym z jego wierzchołków. Rozbudujmy jeszcze trochę naszą terminologię. *Wysokością* wierzchołka v nazwiemy maksimum z odległości między v a jakimś wierzchołkiem w jego poddrzewie.

Synów każdego wierzchołka ustawiamy od lewej do prawej w kolejności nierosnących wysokości. Zaznaczmy, że robimy to tylko na potrzeby omówienia, sam algorytm nie musi korzystać z takiej reprezentacji. Jeżeli v ma synów, to *najstarszym synem* v nazwiemy jego pierwszego syna z lewej (jednego z najwyższych). Powiemy jeszcze, że w jest *pierworodny*, jeżeli jest najstarszym synem swojego ojca. Zauważmy, że korzeń nie jest pierworodny.

Zanim przejdziemy do opisu algorytmu, wprowadzimy kolejny lemat, tym razem o nie całkiem intuicyjnej treści, ale ciekawym dowodzie.

Lemat 4. Niech v będzie wierzchołkiem drzewa T. Przez $pot(v)$ (*potencjał* v) oznaczmy sumę wysokości wszystkich synów v oprócz najstarszego syna. Wówczas suma potencjałów wszystkich wierzchołków drzewa nie przekracza n.

Dowód: Na każdym wierzchołku postawmy jednego krasnoludka, w sumie n krasnoludków.

Następnie dla każdego wierzchołka v wykonujemy co następuje. Niech w będzie pierwszym wierzchołkiem na ścieżce z v do r, który nie jest pierworodny. Posyłamy krasnoludka z v do ojca w, a jeśli $w = r$, to ten krasnoludek opuszcza drzewo.

Pokażemy, że po zakończeniu tej procedury liczba krasnoludków na każdym wierzchołku będzie równa jego potencjałowi. Rozważmy bowiem dowolny wierzchołek v oraz jego syna w, który nie jest najstarszy. Wysokość w jest równa liczbie wierzchołków na ścieżce od w do jego skrajnie lewego liścia. Wszystkie wierzchołki na tej ścieżce oprócz w są pierworodne, czyli z każdego z nich przeszedł do v jeden krasnoludek. Natomiast każdy z pozostałych krasnoludków w poddrzewie w zatrzyma się wcześniej, gdyż na ścieżce od w do niego co najmniej raz nie szliśmy skrajnie w lewo. ■

Ten lemat bardzo przyda się w szacowaniu złożoności czasowej naszego algorytmu.

Podzielmy jeszcze równoodległe trójki w naszym drzewie na dwa rodzaje. Powiemy, że trójka x, y, z jest

- *zgodna*, jeśli wierzchołek spotkania x, y, z jest przodkiem wszystkich trzech wierzchołków,
- *niezgodna* w przeciwnym przypadku; patrz rys. 1.

Rys. 1: Po lewej: zgodna równoodległa trójka. Po prawej: niezgodna równoodległa trójka.

Nasz algorytm będzie dzielił się na dwie fazy:

1. Zliczanie zgodnych równoodległych trójek oraz przygotowanie zapytań związanych ze zliczaniem niezgodnych równoodległych trójek.
2. Przetworzenie zapytań za pomocą centroidalnej dekompozycji.

Faza 1

Będziemy postępować podobnie jak w rozwiązaniu $O(n^2)$, ale skorzystamy z lematu 4, aby złożoność tej fazy była liniowa.

Odrobinę rozszerzymy wprowadzoną wcześniej definicję głębokości. Kiedy v należy do poddrzewa wierzchołka w, to głębokością *v względem w* nazwiemy odległość $d(v, w)$.

Zliczanie równoodległych trójek

Przeszukamy nasze drzewo w głąb za pomocą rekurencyjnej funkcji *dfs*(). Chcemy, aby *dfs*(v) zwracało listę liczb at_depth_v zdefiniowaną w następujący sposób:

- długość at_depth_v równa się wysokości v,
- l-ty element (tym razem indeksujemy od 0) at_depth_v to liczba wierzchołków o głębokości l względem v (czyli zawsze zerowy element to 1).

Na razie skupimy się tylko na tym, później łatwo będzie rozszerzyć funkcję *dfs*(), aby zliczała zgodne równoodległe trójki.

Jeżeli v ma synów, to nie będziemy tworzyć listy-wyniku *dfs*(v) od początku, ale wykorzystamy wynik najstarszego syna v, zaktualizowany o wyniki pozostałych synów.

1. Ponumerujmy synów v jako $v_1, \ldots, v_k$ tak, aby najstarszy miał numer 1. Wywołujemy *dfs*() dla wszystkich synów v i zapisujemy wyniki. Niech $at\_depth_{v_i}$ będzie listą obliczoną dla i-tego syna; dodatkowo oznaczmy $A = at\_depth_{v_1}$.
2. Dodajemy liczbę 1 na początek listy A. Teraz A opisuje głębokości w drzewie składającym się z v i jego najstarszego syna.
3. Dla $i = 2, \ldots, k$ dodajemy elementy listy $at\_depth_{v_i}$ do elementów A (j-ty element $at\_depth_{v_i}$ do $(j+1)$-szego elementu A). Po i-tym kroku lista A opisuje głębokości w drzewie złożonym z v oraz poddrzew synów $v_1, v_2, \ldots, v_i$.
4. Na koniec zwracamy A jako at_depth_v.

Ten opis algorytmu wymaga jeszcze kilku komentarzy.

Po pierwsze, obiecywaliśmy, że nie będziemy korzystać z uporządkowania synów względem nierosnących wysokości. W algorytmie potrzebujemy jedynie stwierdzić, który syn jest najstarszy. Wystarczy wybrać jednego z synów o największej wysokości, tę zaś liczy się bardzo łatwo.

Po drugie, jeśli rozwiązanie piszemy w języku C++, to zamiast listy możemy zwracać dynamicznie alokowaną tablicę – STL-owy `vector`. Trzeba wtedy indeksować elementy od aktualnego końca tablicy, żeby możliwe było dostawienie nowego elementu o indeksie 1. Musimy też uważać, żeby nie kopiować całej tablicy podczas zwracania wyniku (można np. korzystać ze wskaźników).

Na mocy lematu 4 wywołanie *dfs*(r) zajmie $O(n)$ czasu.

Przetwarzając wierzchołek v, chcielibyśmy zliczyć zgodne równoodległe trójki o wierzchołku spotkania v. Robimy to, korzystając z list-wyników dla synów v oraz dwóch dodatkowych zmiennych, tak samo jak w rozwiązaniu $O(n^2)$.

Przygotowanie zapytań dla fazy 2

Chcemy również zrobić coś z niezgodnymi równoodległymi trójkami, dla których v jest wierzchołkiem spotkania. W tej fazie nie wykonamy samego zliczania tych trójek, ale zredukujemy je do innego problemu.

Podczas wywołania funkcji *dfs*(), tworzymy dodatkową listę $pairs\_at\_depth_v$.

> l-ty element $pairs\_at\_depth_v$ to liczba nieuporządkowanych par x, y wierzchołków o głębokości l względem v, należących do poddrzew różnych synów v.

Chcemy przy tym, aby długość tej listy była równa wysokości drugiego najwyższego syna v. Wobec tego możemy obliczać ją dla każdego wierzchołka, nadal zachowując złożoność $O(n)$. Można ją łatwo wyliczyć podczas aktualizowania at_depth_v; szczegóły pozostawiamy Czytelnikowi jako ćwiczenie.

Zobaczmy, jak wygląda niezgodna równoodległa trójka o wierzchołku spotkania v. Dwa wierzchołki należą do poddrzew dwóch różnych synów v i mają tę samą głębokość względem v, powiedzmy l. Trzeci wierzchołek znajduje się poza poddrzewem v i jest odległy od niego o l.

Gdybyśmy potrafili w czasie stałym odpowiadać na następujące pytanie:

„Ile jest wierzchołków odległych od v o zadane l?”,

to wówczas nie mielibyśmy problemu ze zliczeniem tych trójek. Oznaczmy odpowiedź na nasze pytanie przez $at\_distance(v,l)$. Wtedy liczba niezgodnych równoodległych trójek o wierzchołku spotkania v to

$$pairs\_at\_depth_v[l] \cdot (at\_distance(v,l) - at\_depth_v[l]).$$

Nie jesteśmy w stanie obliczać funkcji $at\_distance()$ w czasie stałym. Na szczęście przebieg naszego algorytmu nie zależy od wyników tej funkcji – możemy zapisać wszystkie jej wywołania jako zapytania, odpowiemy na nie zbiorczo w fazie 2. Wraz z zapytaniami zapamiętamy odpowiadające im wartości $pairs\_at\_depth_v[l]$ oraz $at\_depth_v[l]$, aby po przebiegu fazy 2 uaktualnić wynik o odpowiednie iloczyny.

Zauważmy, że w ten sposób wyprodukujemy $O(n)$ zapytań, bowiem suma rozmiarów tablic $pairs\_at\_depth_v$, na mocy lematu 4, szacuje się z góry przez n.

Faza 2

Naszym celem będzie jak najszybsze odpowiedzenie na $O(n)$ zapytań postaci $at\_distance(v,l)$. Wykorzystamy do tego centroidalną dekompozycję drzewa. O samej technice można więcej przeczytać na algorytmicznym blogu prowadzonym przez Pawła Gawrychowskiego[1]. Tutaj przedstawimy ją tylko w zakresie potrzebnym do rozwiązania zadania.

Jeśli po ukorzenieniu T w wierzchołku v poddrzewo każdego syna v ma co najwyżej $\frac{n}{2}$ wierzchołków, to mówimy, że v jest *centroidem* T. Każde drzewo ma jeden lub dwa centroidy. Dowód tego faktu wraz z liniowym algorytmem na znajdowanie centroidów można znaleźć w omówieniu zadania *Polaryzacja* z XX Olimpiady Informatycznej [20].

Nasz algorytm będzie rekurencyjny. Dla każdego zapytania będziemy pamiętać dotychczas obliczoną częściową odpowiedź. Początkowo jest ona równa zero. Chcemy, aby zachodził następujący niezmiennik:

> Częściowa odpowiedź na zapytanie $at\_distance(v,l)$ jest równa liczbie wierzchołków odległych od v o l i znajdujących się w największym dotychczas przetworzonym poddrzewie zawierającym v.

Jeżeli nasz algorytm zostanie wywołany w pewnym momencie dla poddrzewa S naszego wyjściowego drzewa T, to robimy co następuje.

1. Znajdujemy centroid c drzewa S.

2. Usuwamy go. Nasze drzewo S rozspójnia się na poddrzewa $S_1,\ldots,S_k$, które przetwarzamy rekurencyjnie.

3. Przeszukujemy drzewo S z wierzchołka c. Dla każdego poddrzewa S_i obliczamy dwie tablice:

    - $at\_depth_i[]$ – gdzie $at\_depth_i[l]$ to liczba wierzchołków odległych od c o l w poddrzewie S_i,

[1] `http://fajnezadania.wordpress.com/2013/02/19/ile-sciezek/`

- $queries_i[]$ – wszystkie zapytania w poddrzewie S_i.

A także jedną tablicę $at\_depth_c[]$, gdzie $at\_depth_c[l]$ to liczba wierzchołków w całym drzewie S odległych od c o l. Ponadto dla każdego wierzchołka obliczamy jego odległość od c.

4. Korzystając z tablicy $at\_depth_c[]$, uaktualniamy odpowiedzi na zapytania postaci $at\_distance(c, \ldots)$.

5. Następnie dla każdego drzewa S_i i zapytania postaci $at\_distance(v, l)$, $v \in S_i$, zliczamy wierzchołki odległe od v o l, które należą do S, a nie należą do S_i. Ta liczba to po prostu:

$$at\_depth_c[l - d(c, v)] - at\_depth_i[l - d(c, v)].$$

Zwiększamy aktualną odpowiedź na to zapytanie o powyższą wartość.

Wywołujemy nasz algorytm dla całego drzewa T. Nasz niezmiennik gwarantuje nam, że uzyskamy w ten sposób pełne odpowiedzi na wszystkie zapytania.

Pozostaje pytanie o złożoność czasową naszego programu. Skorzystamy z argumentu „obliczeń na kolejnych poziomach", który pojawia się często w dowodzie złożoności sortowania przez scalanie. Będziemy rozważać następujące poziomy:

- całe drzewo T,
- poddrzewa powstałe z T po usunięciu jego centroidu,
- poddrzewa powstałe po usunięciu ich centroidów,
- ...
- pojedyncze wierzchołki.

Na każdym z poziomów mamy w sumie n wierzchołków oraz $O(n)$ zapytań dotyczących tych wierzchołków. Nasz algorytm zużywa zatem $O(n)$ czasu na poziom. Poziomów może być co najwyżej $\log n$, ponieważ usuwając centroid, dwukrotnie zmniejszamy rozmiar największego drzewa.

Zakończyliśmy w ten sposób fazę 2, a tym samym cały algorytm. Jego czas działania to $O(n + n \log n) = O(n \log n)$. Został zaimplementowany w pliku `hot4.cpp`.

Czytelników zainteresowanych metodą centroidalnej dekompozycji zachęcamy do zmierzenia się z zadaniem *Këbab* z XIV Obozu im. A. Kreczmara. Jest ono dostępne w serwisie `main.edu.pl`.

Testy

Przygotowano dziesięć grup testów. W pierwszej grupie znajdują się małe testy – od 1 do 5 wierzchołków. W każdej z pozostałych grup znajdują się po dwa testy. Testy typu *a* są losowe, zaś w testach typu *b* jeden z wierzchołków ma zagwarantowany duży stopień.

Test *10b* to gwiazda złożona z 5000 wierzchołków – test z największym możliwym do uzyskania wynikiem.