Wojciech Nadara — Treść zadania, Opracowanie
Jan Kanty Milczek — Program

Dostępna pamięć: 256 MB. *OI, etap III, dzień próbny, 15.04.2015*

Odwiedziny

Bajtazar to wyjątkowy człowiek – przez ostatnie 21 lat był listonoszem, bankierem, łyżwiarzem, a nawet królem! Nic dziwnego, że ma mnóstwo znajomych. Niestety, ciągłe zmiany miejsc pracy sprawiły, że z wieloma z nich zaczął tracić kontakt... Czas to zmienić! Bajtazar wykona wielkie tournée po Bajtocji, aby odnowić stare znajomości.

W Bajtocji znajduje się n miast, połączonych siecią $n-1$ dwukierunkowych dróg. Nasz bohater chce odwiedzić każde z miast kraju i ustalił już konkretną kolejność wizyt. Trasę między każdymi dwoma kolejnymi miastami pokona, korzystając z samochodu wypożyczonego w BMW (Bajtockiej Motoryzacji Wycieczkowej). Wypożyczenie każdego samochodu nie kosztuje nic, ale auta trzeba tankować – samochód o pojemności baku k trzeba zatankować w mieście początkowym trasy i każdorazowo po przejechaniu dokładnie k dróg. BMW, znając plan tournée Bajtazara oraz wiedząc, że każdą trasę będzie on chciał pokonać jak najszybciej, tak dobrało pojemność baków wypożyczanych samochodów, aby musiał on każdy z nich zatankować również w mieście docelowym.

Znając kolejność, w jakiej Bajtazar odwiedzi miasta, ceny tankowania w każdym z nich oraz pojemności baków wypożyczanych samochodów, wyznacz, ile będzie go kosztowało przejechanie każdej trasy.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($2 \leqslant n \leqslant 50\,000$), oznaczająca liczbę miast w Bajtocji. Miasta są numerowane od 1 do n. W kolejnym wierszu znajduje się ciąg n liczb całkowitych $c_1, \ldots, c_n$ ($1 \leqslant c_i \leqslant 10\,000$) pooddzielanych pojedynczymi odstępami, oznaczających ceny paliwa w miastach Bajtocji: liczba c_i oznacza koszt napełnienia baku dowolnego samochodu w mieście o numerze i.

Dalej następuje $n-1$ wierszy zawierających opisy dróg w Bajtocji. W każdym z nich podane są dwie liczby całkowite a, b ($1 \leqslant a, b \leqslant n$) oddzielone pojedynczym odstępem, oznaczające, że w Bajtocji istnieje dwukierunkowa droga łącząca miasta o numerach a i b.

W następnym wierszu znajduje się ciąg n liczb całkowitych $t_1, \ldots, t_n$ pooddzielanych pojedynczymi odstępami, opisujący kolejność, w jakiej Bajtazar ma zamiar odwiedzić miasta (każda z liczb od 1 do n pojawi się w tym ciągu dokładnie raz). Ostatni wiersz wejścia zawiera ciąg $n-1$ liczb całkowitych $k_1, \ldots, k_{n-1}$ pooddzielanych pojedynczymi odstępami, opisujący pojemności baków wypożyczanych samochodów: liczba k_i oznacza, że podczas przejazdu z miasta o numerze t_i do miasta o numerze t_{i+1}, Bajtazar będzie musiał tankować samochód co k_i dróg. Możesz założyć, że k_i zawsze dzieli odległość między tymi miastami.

W testach wartych 28% punktów zachodzi dodatkowy warunek $n \leqslant 1000$, a w testach wartych 36% punktów zachodzi dodatkowy warunek $n \leqslant 10\,000$.

Wyjście

Na standardowe wyjście Twój program powinien wypisać $n-1$ wierszy, w każdym po jednej liczbie całkowitej. Liczba w i-tym wierszu ma oznaczać łączny koszt tankowania podczas trasy z miasta o numerze t_i do miasta o numerze t_{i+1}.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
5	10
1 2 3 4 5	6
1 2	10
2 3	5
3 4	
3 5	
4 1 5 2 3	
1 3 1 1	

Rozwiązanie

Przedstawimy rozwiązanie nieco ogólniejszego problemu. Dane jest drzewo o n wierzchołkach (czyli spójny graf o n wierzchołkach i $n-1$ krawędziach). Każdy wierzchołek ma przypisaną wagę. Dodatkowo dana jest również lista q *zapytań*. Każde zapytanie opisane jest za pomocą pary wierzchołków u i v oraz liczby k. Takie zapytanie polega na obliczeniu sumy wag wierzchołków leżących na ścieżce łączącej u i v, przy czym do sumy wliczamy jedynie wagi co k-tego wierzchołka. Możemy przy tym założyć, że odległość między u i v jest podzielna przez k. W dalszym opisie załóżmy, że i-te zapytanie dane jest za pomocą trzech parametrów: u_i, v_i oraz k_i.

Rozwiązanie brutalne

Zacznijmy od dosyć bezpośredniego rozwiązania. Zauważmy, że dla ustalonego k, w algorytmie przeszukiwania w głąb (DFS) możemy bez trudu utrzymywać sumę wag co k-tego wierzchołka na ścieżce od korzenia do aktualnie przetwarzanego wierzchołka. Dla każdego z q zapytań uruchamiamy więc algorytm DFS startujący w u_i i gdy tylko dotrzemy do v_i (który leży na głębokości podzielnej przez k), możemy odczytać w nim wynik zapytania. To bardzo proste rozwiązanie działa w złożoności czasowej $O(nq)$ i pamięciowej $O(n)$, jednak niestety nie przydaje się zbytnio na drodze do szybszego rozwiązania.

Początkowe przemyślenia

Zadanie jest daleko idącym uogólnieniem klasycznego problemu, w którym dany jest ciąg liczb, a naszym celem jest odpowiadanie na zapytania dotyczące sumy liczb

z podanego fragmentu tego ciągu. Problem ten ma dwa standardowe rozwiązania. Pierwsze z nich opiera się na obliczeniu sum prefiksowych ciągu, natomiast drugie korzysta z drzewa przedziałowego. Drugie z tych rozwiązań jest istotnie bardziej skomplikowane oraz wolniejsze. Warto je stosować dopiero w przypadku, gdy w oryginalnym problemie zostały wprowadzone pewne utrudnienia, na przykład liczby w danym ciągu mogą się zmieniać w czasie. W naszym zadaniu musimy poradzić sobie z dwiema przeszkodami. Po pierwsze, rozważamy dowolne drzewo, a nie zwykły ciąg (który odpowiadałby bardzo specyficznemu drzewu). Po drugie, sumujemy wagi nie każdego, lecz co k-tego wierzchołka.

Aby dojść do rozwiązania naszego zadania, zastanówmy się, jak wcześniej wspomniane metody radzą sobie z tymi utrudnieniami. Po chwili namysłu można dojść do wniosku, że drzewo przedziałowe w przypadku naszego zadania nie ma przewagi nad sumami prefiksowymi – jego główna zaleta (możliwość modyfikacji elementów w czasie) nie przydaje się. Możemy zatem zastanowić się, jak zmodyfikować rozwiązanie używające sum prefiksowych, aby stosowało się do naszego zadania.

Zajmijmy się na chwilę jedynie zapytaniami z $k = 1$. Ukorzeńmy rozważane drzewo w dowolnym wierzchołku. Wówczas ścieżkę $u - v$ między dowolnymi dwoma wierzchołkami u i v można rozbić na dwie ścieżki $u - s$ i $s - v$, gdzie s jest *najniższym wspólnym przodkiem* wierzchołków u i v (ang. *lowest common ancestor*, *LCA*). Co więcej, ścieżki te biegną od u i v cały czas w stronę korzenia. Dla wygody oznaczmy przez $s = LCA(u, v)$ najniższego wspólnego przodka wierzchołków u i v. Teraz korzystamy z techniki sum prefiksowych. Dla każdego wierzchołka v utrzymujemy sumę wag $suma[v]$ wierzchołków na ścieżce od v do korzenia drzewa. Aby wyznaczyć sumę wag wierzchołków na ścieżce od u do v, wystarczy teraz obliczyć $suma[u] + suma[v] - 2 \cdot suma[LCA(u, v)] + waga[LCA(u, v)]$.

Ten pomysł, połączony ze strukturą danych pozwalającą efektywnie obliczać $LCA(u, v)$, pozwala rozwiązać zadanie w przypadku $k = 1$. Całkiem dobrze poradziliśmy sobie z faktem, że musimy pracować z drzewem a nie ciągiem liczb. Jednak aby rozwiązać zadanie dla dowolnych wartości k, będziemy potrzebowali całkiem nowego pomysłu.

Rozwiązanie wzorcowe

W naszym rozwiązaniu wielokrotnie będziemy potrzebować odpowiedzi na zapytania postaci „Czy wierzchołek a jest przodkiem wierzchołka b?”. Problem ten daje się rozwiązać dosyć standardową techniką. Uruchamiamy algorytm DFS i dla każdego wierzchołka zapamiętujemy czas wejścia i wyjścia. Znając te wartości, na wspomniane zapytania możemy odpowiadać w czasie stałym.

Jak już wspomnieliśmy, będziemy operowali na ścieżkach w drzewie, dlatego przyda nam się także struktura, która pozwala odpowiadać na zapytania o najniższego wspólnego przodka dowolnych dwóch wierzchołków. Można ją zaimplementować na kilka różnych sposobów, jednak, jak się za chwilę okaże, w naszym przypadku jeden z nich będzie szczególnie wygodny.

Podzielmy zapytania na dwie grupy: te z dużymi parametrami k_i (dalej zwane *dużymi*) i te z małymi parametrami k_i (dalej zwane *małymi*). Parametr k_i uznajemy

za duży, gdy $k_i \geqslant t$, przy czym wartość t dobierzemy później. Odpowiedzi na duże i małe zapytania będziemy uzyskiwać na różne sposoby.

Duże zapytania

Zajmijmy się najpierw dużymi zapytaniami. Zauważmy, że jeżeli $k_i \geqslant t$, to poszukiwana suma składa się z co najwyżej $\frac{n}{t} + 1$ wag wierzchołków. Zastosujemy podejście brutalne: składniki będziemy dodawać po jednym. Jak już wspomnieliśmy, każdą ścieżkę w ukorzenionym drzewie między dwoma wierzchołkami u_i oraz v_i można rozbić na dwie ścieżki: od u_i do $LCA(u_i, v_i)$ oraz od $LCA(u_i, v_i)$ do v_i (potencjalnie któraś z tych części składa się z tylko jednego wierzchołka). Każdą taką ścieżką zajmiemy się osobno. Wierzchołki, które musimy wliczyć do sumy na takiej ścieżce, można uzyskać w taki sposób, że zaczynamy od (dla ustalenia uwagi) u_i i skaczemy co k_i wierzchołków w górę drzewa, do momentu, w którym znajdziemy się powyżej $LCA(u_i, v_i)$. To, czy przeskoczyliśmy $LCA(u_i, v_i)$, czy nie, sprawdzamy, pytając, czy nasz aktualny wierzchołek jest przodkiem v_i. Pozostaje nam rozwiązać następujący problem: jak odpowiadać na zapytania postaci „Jaki jest wierzchołek o k w górę od wierzchołka v?". Standardowe rozwiązanie korzysta z tzw. *jump pointers*, których używa się w jednym ze sposobów obliczania LCA, jednak ich wykorzystanie spowoduje, że na takie pytania będziemy odpowiadać w czasie $O(\log k)$. Zastosujemy zatem inne, szybsze podejście.

Dla każdego z dużych zapytań u_i, v_i, k_i zapiszmy w wierzchołkach drzewa u_i oraz v_i, że należą one do i-tego zapytania. Ukorzeńmy drzewo w dowolnym wierzchołku. Następnie uruchommy algorytm DFS, który w pomocniczej tablicy utrzymuje numery wierzchołków na ścieżce od korzenia do aktualnie rozpatrywanego wierzchołka. Aktualizacja tej tablicy przy zejściu w głąb wymaga jedynie dopisania jednego numeru wierzchołka na końcu tablicy. Z kolei po powrocie z wywołania rekurencyjnego wystarczy usunąć z niej ostatni element. Mając taką tablicę pomocniczą, w dowolnym momencie, dla dowolnego parametru k możemy odczytać z niej numer wierzchołka, który leży dokładnie k poziomów powyżej aktualnie rozpatrywanego wierzchołka. Taką pomocniczą tablicę można zaimplementować za pomocą zwykłej tablicy pamiętającej indeks ostatniego elementu lub struktury `vector` z C++.

Pokażemy teraz, jak to wykorzystać w naszym algorytmie. W momencie, gdy przeszukiwanie *powraca* z wierzchołka u_i (lub odpowiednio v_i), dodajmy wagę wierzchołka u_i (lub v_i) do licznika odpowiadającego za odpowiedź na i-te zapytanie. Następnie w wierzchołku, który jest o k w górę od u_i (dostęp do takiego mamy teraz w czasie stałym!), zapiszmy, że mamy go później przetworzyć w taki sam sposób, w jaki przetworzyliśmy u_i (o ile jeszcze nie leży on powyżej $LCA(u_i, v_i)$). W taki sposób na każdy z wierzchołków, które musimy wliczyć do odpowiedzi, poświęcamy stałą liczbę operacji, zatem odpowiedzi na wszystkie duże zapytania jesteśmy w stanie wyznaczyć w czasie $O(\frac{nq}{t})$. Należy przy tym uważać na mały szczegół techniczny: łatwo pomylić się i wagę LCA policzyć w tej sumie dwukrotnie.

Warto tu jeszcze zwrócić uwagę na złożoność pamięciową. Każde duże zapytanie w dowolnym momencie wykonania algorytmu odpowiada za istnienie co najwyżej dwóch elementów na listach zapytań, które utrzymujemy w wierzchołkach. Łącznie daje to $O(q)$ elementów. Jeśli jednak implementujemy algorytm w języku C++ i do przechowywania tych list używamy struktur `vector`, możemy wpaść w pułapkę. Struktura `vector` nie gwarantuje, że zajmuje pamięć proporcjonalną do długości *aktualnej* listy

elementów. Pesymistycznie może zajmować pamięć proporcjonalną do maksymalnego *dotychczas* osiągniętego rozmiaru. To powoduje, że złożoność pamięciowa naszego programu niepotrzebnie wzrasta do $O(\frac{nq}{t})$. Aby pozbyć się tego problemu, po przetworzeniu listy z danego wierzchołka powinniśmy wyczyścić wektor i wywołać metodę `shrink_to_fit`, która powoduje, że wektor zwalnia nieużywaną pamięć. To poprawia złożoność pamięciową do $O(q)$.

Małe zapytania

Zajmijmy się teraz małymi zapytaniami. Tu przyda nam się idea sum prefiksowych. Załóżmy, że dla każdego $k < t$ i każdego wierzchołka v znamy sumę wag co k-tego wierzchołka na ścieżce do korzenia (sumę prefiksową). Oznaczmy tę wartość przez $suma[v][k]$. Wówczas odpowiedź na zapytanie możemy wyznaczyć następująco. Ponownie rozdzielamy ścieżkę z u_i do v_i na dwie ścieżki od u_i i v_i do $LCA(u_i, v_i)$ i następnie obliczamy sumę odpowiednich wag na każdej z nich za pomocą uprzednio spamiętanych sum prefiksowych. Trzeba przy tym uważać, aby wagi $LCA(u_i, v_i)$ nie policzyć dwa razy.

Aby obliczyć sumę wag na ścieżce od u_i do $LCA(u_i, v_i)$, znajdujemy najniższego przodka u_i, który leży powyżej $LCA(u_i, v_i)$ i którego odległość od u_i jest podzielna przez k_i. Oznaczmy znaleziony wierzchołek przez w. Wówczas suma wag na ścieżce od u_i do $LCA(u_i, v_i)$ to $suma[u_i][k_i] - suma[w][k_i]$ (jeśli w nie istnieje, szukana wartość to $suma[u_i][k_i]$). Zauważmy, że ponownie pojawia się tutaj pytanie o przodka ustalonego wierzchołka na danej głębokości, zatem i tu posłużymy się algorytmem DFS z pomocniczą tablicą. Jeśli znamy $LCA(u_i, v_i)$ i głębokości wierzchołków, wyznaczenie takiego w nie stanowi problemu. Nasze zadanie sprowadziliśmy zatem do problemu obliczenia tablicy $suma$, co już jest dosyć proste. Uruchamiamy algorytm DFS z pomocniczą tablicą i przy rozpatrywaniu wierzchołka v dla wszystkich liczb k z przedziału $[1, t)$ uaktualniamy $suma[v][k] := waga[v] + suma[s][k]$, gdzie s jest przodkiem v odległym o k (jeśli taki wierzchołek s nie istnieje, wówczas po prostu przyjmujemy $suma[v][k] = waga[v]$).

Takim oto sposobem uzyskaliśmy rozwiązanie tej części zadania w złożoności czasowej $O(nt+q \log n)$ i pamięciowej $O(nt)$. Składnik $O(q \log n)$ pochodzi z wyznaczania LCA – o czym piszemy w następnej sekcji.

Wyznaczanie LCA

Jak już się przekonaliśmy, w całym naszym rozwiązaniu wykorzystujemy algorytm DFS pamiętający ścieżkę od korzenia. Jeśli mamy tę ścieżkę oraz funkcję odpowiadającą w czasie stałym na pytanie „Czy a jest przodkiem b?", wyznaczenie $LCA(u, v)$ jest bardzo proste – w momencie, gdy algorytm DFS odwiedza wierzchołek u, wystarczy na ścieżce od korzenia do u znaleźć najniższy wierzchołek, który jest przodkiem v. Ścieżkę od korzenia do u mamy wówczas zapisaną w tablicy, dlatego możemy w tym celu użyć wyszukiwania binarnego. Co ciekawe, jeżeli przyjrzymy się bliżej naszemu algorytmowi, okaże się, że nigdy nie potrzebujemy obliczać samego LCA. W przypadku dużych zapytań skakaliśmy co k, aż znaleźliśmy się w przodku drugiego wierzchołka z zapytania, a w przypadku małych możemy bezpośrednio od razu znaleźć w, szukając

najniższego wierzchołka będącego przodkiem v_i (którego poprzednik na ścieżce nie jest przodkiem v_i) i rozpatrując jedynie przodków u_i w odległości podzielnej przez k_i.

Dobór parametru t

Umiemy już w miarę szybko odpowiadać zarówno na duże jak i na małe zapytania. Pozostało jedynie dobrać optymalną wartość parametru t. Na duże zapytania odpowiadamy w sumarycznym czasie $O(\frac{nq}{t})$, a na małe w czasie $O(nt + q \log n)$, przez co optymalna wartość t wynosi $\sqrt{q}$. Wówczas otrzymujemy rozwiązanie o złożoności czasowej $O(n\sqrt{q} + q \log n)$ oraz złożoności pamięciowej $O(n\sqrt{q})$. Doświadczeni zawodnicy pewnie od początku przeczuwali, że w złożoności pojawi się pierwiastek z q lub z n. W takich sytuacjach warto jednak nie polegać ślepo na intuicji i przeprowadzić dokładną analizę w zależności od wprowadzonego parametru. Może się na przykład zdarzyć, że czasy działania dwóch faz to, powiedzmy, $O(nt)$ i $O(\frac{n}{t} \log n)$. Wtedy dla $t = \sqrt{n}$ czas działania całego algorytmu to $O(n\sqrt{n} \log n)$, jednak dla $t = \sqrt{n \log n}$ cały algorytm działa w czasie $O(n\sqrt{n \log n})$, czyli lepszym o czynnik $\sqrt{\log n}$. Taka różnica może być kluczowa do otrzymania kompletu punktów za rozwiązanie.

Redukcja złożoności pamięciowej

Warto wspomnieć, że złożoność pamięciową można zmniejszyć do liniowej, choć to usprawnienie nie było wymagane od zawodników. Opisaliśmy już, jak ustrzec się przed zwiększonym zużyciem pamięci w przypadku rozpatrywania dużych zapytań, zatem czemu nie spróbować zredukować jej w przypadku małych zapytań? Aby odpowiadać na małe zapytania, najpierw wykonaliśmy kosztowny preprocessing w czasie $O(n\sqrt{q})$, a potem raz przetworzyliśmy zapytania w czasie $O(q \log n + n)$, aby uzyskać wszystkie odpowiedzi. Nie trzeba jednak deklarować od razu wielkiej tablicy *suma* zajmującej $O(n\sqrt{q})$ pamięci. Cały algorytm można podzielić na t faz, gdzie w p-tej fazie odpowiadamy na zapytania, w których $k_i = p$. Przetwarzając wszystkie zapytania z ustalonym k, potrzebujemy jedynie sum prefiksowych skaczących co k, co redukuje pamięć do $O(n)$. Ostatecznie otrzymujemy algorytm rozwiązujący zadanie w złożoności czasowej $O(n\sqrt{q} + q \log n)$ i pamięciowej $O(n + q)$.

Możliwe udogodnienia implementacyjne

W rozwiązaniu wzorcowym wielokrotnie używaliśmy zapytań postaci „Jaki jest przodek wierzchołka v w odległości k?", na które odpowiadaliśmy za pomocą algorytmu DFS z pomocniczą tablicą. Wymagało to uprzedniego wczytania wszystkich zapytań i dzielenia algorytmu na poszczególne fazy. Co zrobić, aby na wspomniane wyżej zapytania odpowiadać *online*? Standardowym rozwiązaniem tego problemu jest metoda *jump pointers*, która jest chyba najpopularniejszym sposobem obliczania LCA. Na początku dla każdej pary (v, d) wyznaczamy przodka v w odległości 2^d (ograniczamy się do takich d, że v leży na głębokości co najmniej 2^d). Zauważmy, że wyniki dla par $(\cdot, d+1)$ łatwo obliczyć na podstawie wyników dla par $(\cdot, d)$. Mając takie informacje, jesteśmy w stanie odpowiadać na pytania postaci „Jaki jest przodek wierzchołka v

w odległości k?" w złożoności czasowej $O(\log n)$, zapisując k w systemie binarnym. Takie rozwiązanie skutkuje zwiększeniem złożoności pamięciowej do $O(n \log n)$ (co jednak nie jest wielką przeszkodą, bo nawet $O(n\sqrt{n})$ było dopuszczane), ale także złożoności czasowej do $O(n\sqrt{q} \log n)$. Jednakże limity czasu i pamięci były na tyle duże, że i takie rozwiązania dostawały na zawodach komplet punktów.

Rozwiązanie alternatywne

Opiszemy jeszcze stosunkowo proste i bardzo pomysłowe rozwiązanie alternatywne. Wcześniej rozpatrywaliśmy tablicę *suma* o wymiarach $n \times \sqrt{q}$, gdyż ograniczyliśmy się do wartości $k < \sqrt{q}$. Teraz pozbądźmy się tego ograniczenia i wyobraźmy sobie, że używamy analogicznej tablicy o wymiarach $n \times n$. Wartości tej tablicy będziemy wyznaczali jedynie tam, gdzie jest to potrzebne. Nie będziemy także oczywiście tak wielkiej tablicy deklarowali jawnie. Zamiast tego posłużymy się słownikową strukturą danych. W C++ najlepiej do tego celu nadaje się `unordered_map`, będąca implementacją tablicy z haszowaniem.

Zdefiniujmy funkcję $F(a, k)$, która oblicza to samo, co w poprzednim rozwiązaniu znaczyło $suma[a][k]$, czyli sumę wag co k-tych wierzchołków na drodze do korzenia, zaczynając od wierzchołka a. Obliczmy $F(a, k)$ za pomocą następującego algorytmu: jeżeli już kiedyś obliczyliśmy $F(a, k)$, to podajemy zapamiętany wynik, a jeżeli nie, to wynikiem jest $waga[a] + F(b, k)$, gdzie $waga[a]$ to oczywiście waga wierzchołka a, a b jest k-tym przodkiem a (o ile istnieje). Zauważmy, że w sumie w całym naszym programie funkcja F nie zostanie wywołana więcej niż $O(q + n\sqrt{q})$ razy! Dzieje się tak z tego samego powodu, z którego poprzednie rozwiązanie działało szybko, tzn. oddzielnie szacujemy liczbę wywołań F dla małych i dużych zapytań. Zaletą takiego podejścia jest jednak to, że podział na małe i duże zapytania odbywa się jedynie w analizie złożoności czasowej (i pamięciowej) i nie musimy się nim przejmować w implementacji.

Wyznaczanie przodka w odległości k możemy zrealizować albo w czasie logarytmicznym za pomocą wcześniej opisanego sposobu, albo w czasie stałym przy użyciu algorytmu DFS z pomocniczą tablicą. Struktura `unordered_map` działa z kolei w oczekiwanym czasie stałym. Zatem w ten sposób możemy otrzymać rozwiązanie w złożoności czasowej $O(n\sqrt{q})$. Jego jedyną wadą w porównaniu do poprzedniego jest gorsza złożoność pamięciowa – $O(n\sqrt{q})$. Oczywiście trzeba tu jeszcze zadbać o kilka kwestii podobnie jak w poprzednim rozwiązaniu, np. wyznaczanie pierwszego wierzchołka ponad LCA w odległości podzielnej przez k.