AISD lista 3

Dominik Szczepaniak

August 29, 2024

1 Zadanie 1

Mamy:

$$T(n) = \begin{cases} 1 & \text{dla } n = 1 \\ 2 * T(n/2) + n/logn & \text{dla } n > 1 \end{cases}$$

Używając metody drzewa rekursji, możemy zauważyć, że na każdym poziomie rekursji mamy n dwukrotnie więcej niż na poprzednim poziomie, a także pojawia się dodatkowo wyrażenie $\frac{n}{\log \frac{n}{2^k}}$, gdzie k to numer poziomu rekursji.

Możemy przyjąć, że głębokość rekursji wynosi $\log n$, ponieważ za każdym razem dzielimy n przez 2, aż do osiągnięcia 1.

Oszacujmy sumę wyrażeń $\frac{n}{\log \frac{n}{2^k}}$. Wartość k będzie od 0 do $\log n$, ponieważ dzielimy n na 2^k części, aż do osiągnięcia 1.

$\sum_{k=0}^{\log n} \frac{n}{\log \frac{n}{2^k}} \leq n \sum_{k=0}^{\log n} \frac{1}{\log \frac{n}{2^k}}$

Możemy przyjąć, że suma jest mniejsza lub równa n razy maksymalna wartość z całego wyrażenia. Maksymalna wartość tego wyrażenia wystąpi, gdy $k = 0$, ponieważ wtedy mamy najmniejszy mianownik.

$\sum_{k=0}^{\log n} \frac{1}{\log \frac{n}{2^k}} \leq \log n \cdot \frac{1}{\log \frac{n}{2^0}} = \log n \cdot \frac{1}{\log n} = 1$

Więc:

$\sum_{k=0}^{\log n} \frac{n}{\log \frac{n}{2^k}} \leq n \cdot 1 = n$

Ostatecznie, $T(n) = \Theta(n)$.

2 Zadanie 2

Idea:

sortujemy po a

trzymamy stos (punkt przeciecia z poprzednia, linia)

nie wrzucamy pierwszej bo ja zawsze widac, wrzucamy druga i liczymy punkt przeciecia

dodajemy trzecia. musimy teraz sprawdzic gdzie przecina sie z druga.

niech X bedzie punktem przeciecia z druga linia, a Y niech bedzie punktem przeciecia drugiej z pierwsza

jesli X $<$ Y to znaczy ze ta linia zakrywa druga poniewaz nad tym punktem przeciecia jest linia pierwsza (poniewaz linia druga z pierwsza przeciela sie dalej), w takim razie mozemy usunac druga ze stosu

jesli X $>$ Y to znaczy ze linia trzecia i druga sa widoczne

ogolny przypadek:

dopoki punkt przeciecia dodawanej linii i linii ze stosu jest na lewo od punktu przeciecia poprzedniej linii z poprzednia (czyli punktu przeciecia ze stosu) to usuwamy linie ze stosu

Algo:

```
def main(proste):
        proste.sort_po_a()
        stos = Stack()
        stos.push({punkt_przeciecia(proste[0], proste[1]), proste[1]})
        for i in range(2, len(proste)):
                while(stos.top()[0] > punkt_przeciecia(proste[i], stos.t
                        stop.pop()
                stos.push({punkt_przeciecia(proste[i], stos.top()[1]), p

        while(!stos.empty()){
                print(stos.top()[1])
                stos.pop()
        }
        print(proste[0])
```

Złożoność:

O(nlogn) - sort

Gdy dodajemy linie na stos to dodamy ja tylko raz. Gdy usuniemy linie ze stosu to usuniemy ja tylko raz. W takim razie mamy O(2n)

Czyli O(nlogn)

Pamięciowa:

O(n) - tablica z prostymi + stos

Dowód poprawności:

Załóżmy nie wprost że powyższy algorytm jest niepoprawny i że istnieje jakaś prosta która jest widoczna w poprawnym rozwiązaniu, a nie jest widoczna w naszym rozwiązaniu. W takim razie jeśli prosta ma być widoczna a nie jest, to istnieje jakaś prosta która przykryła ją w jakimś momencie dodawania jej do stosu. No ale przy dodawaniu do stosu sprawdzamy czy punkt przecięcia prostej którą mamy usunąć do poprzedniej prostej jest na prawo od przecięcia obecnie rozpatrywanej prostej z prostą która ma być usunięta. Usuwamy tylko jeśli jest na prawo, więc tak musiało być. No ale jeśli przecięcie jest na prawo, to ponieważ ta prosta rośnie szybciej niż prosta wcześniej rozpatrywana (bo sortowaliśmy po rosnącym a), to każdy w każdym punkcie na prawo od przecięcia tych dwóch prostych prosta szybciej rosnąca będzie miała większą wartość (będzie wyżej). W takim razie jeśli usuwana prosta zaczyna być widoczna dopiero w punkcie który na prawo od przecięcia z obecnie rozpatrywaną prostą to nie jest nigdy widoczna, bo obecnie rozpatrywana prosta zakrywa ją. W takim razie doszliśmy do sprzeczności że usunięta prosta miałaby być widoczna, więc algorytm musi działać poprawnie.

3 Zadanie 3

a)

Niech nasza liczba A = $(a_n, a_{n-1}, ..., a_1, a_0)_r$ - liczba z n+1 liczbami o podstawie r.

Wynik: Liczba C = A*A = A^2

Algorytm:

1. Jeśli n = 1 zwróć A * A = A^2

2. Podzielmy A na dwie równe części $A_L i A_R$:

$A = A_L * r^{n/2} + A_R$

3. Obliczmy:

$d_1 = reku(A_L)$

$d_0 = reku(A_R)$

$d_{0,1} = reku(A_L + A_R)$

4. Zwróćmy:

C = $d_1 * r^n + (d_{0,1} - d_0 - d_1) * r^{n/2} + d_0$

$A * A = (A_L * r^{n/2} + A_R)^2 = (A_L^2 * r^n + 2 * A_L * r^{n/2} * A_R + A_R^2)$

Niech $d_0 = A_L^2$

Niech $d_1 = A_R^2$

Niech $d_{0,1} = (A_L + A_R)^2$

Czyli $d_{0,1} - d_0^2 - d_1^2 = 2 * A_L * A_R * r^{n/2}$

W takim razie możemy iść rekurencyjnie i dostaniemy poprawny wynik.

Zlozonosc taka sama jak podstawowy karatsuba, bo robi dokladnie to samo przeciez:

$O(n^{log_2 3})$

b)

Liczba a $= a_0 * x^2 + a_1 * x + a_0 * x^0$

$x = 10^{\frac{n}{3}}$

Nazwijmy $a_0 = a, a_1 = b, a_2 = c$

Wtedy a * a $= a * a * x^4 + 2ab * x^3 + (2ac + b^2) * x^2 + 2bc * x + c^2$

Niech

$a^2 = C_4$

$2ab = C_3$

$2ac + b^2 = C_2$

$2bc = C_1$

$c^2 = C_0$

Wtedy nasz algorytm:

$X_0 = C_0 = c^2$

$X_1 = (C_4 + C_3 + C_2 + C_1 + C_0) = (a + b + c)^2$

$X_2 = (C_4 - C_3 + C_2 - C_1 + C_0) = (a - b + c)^2$

$X_3 = (16C_4 + 8C_3 + 4C_2 + 2C_1 + C_0) = (4a + 2b + c)^2$

$X_4 = C_4 = a^2$

X_4 oraz X_0 liczymy rekurencyjnie wywołując nasz algorytm na odpowiednio c^2 oraz a^2

$X_1 - X_2 = 2(C_4 + C_2 + C_0)$

Znamy już C_4 oraz C_0, więc:

$C_2 = \frac{X_1 - X_2}{2} - C_4 - C_0$

Dzielenie przez 2 robimy brute-forcując

Jak bruteforce:

Mamy liczbe 12345678

liczymy:
1 / 2 < 1
12 / 2 = 6
3 / 2 = 1 r 1
4+10 / 2 = 7
5 / 2 = 2 r 1
16 / 2 = 8
7 / 2 = 3 r 1
18 / 2 = 9
Liczba:
6172839

Jak na końcu zostanie reszta 1 to ją ignorujemy - wtedy liczba była nieparzysta.

$X_3 - 2X_1 = 14C_4 + 6C_3 + 2C_2 - C_0$
Znamy $C_4, C_2 i C_0$, więc z tego mamy C_3.
14 = (10+4)
6 = (10-4)
$10(C_4 + C_3) + 4(C_4 - C_3)$
10 to dodanie zera na koniec.

Mnożenie przez 14, 6 i 2 wykonujemy standardowo bruteforcując Te liczby mają krótki zapis binarny - do 5 cyfr, więc nie jest to problemem przy dużych liczbach.

$X_1 - X_2 - X_3 + 2X_4 = C_1$

Złożoność $T(n) = 5T(n/3) + O(n) = O(n^{log_3(5)})$

Dla ogólnego przypadku mamy:
$(a1 * 10^{\frac{(k-1)*n}{k}} + a2 * 10^{\frac{(k-2)*n}{k}} + ... + a_k * 1)^2 =$
$(a*10^{3n/4}+b*10^{2n/4}+c*10^{n/4}+d)^2 = a^2*10^{6n/4}+2ab*10^{5n/4}+2ac*10^n+ 2ad*10^{3n/4}+b^2*10^n+2bc*10^{3n/4}+2bd*10^{2n/4}+c^2*10^{2n/4}+2cd*10^{n/4}+d^2 = a^2*10^{6n/4}+2ab*10^{5n/4}+10^n(2ac+b^2)+10^{3n/4}(2ad+2bc)+10^{2n/4}(2bd+ c^2)+2cd*10^{n/4}+d^2$

Ogólnie mamy liczbe:
$a * x^{(k-1)n/k} + b * x^{(k-2)n/k} + ... + z * x^0$

Nasze mnożenie to mnożenie dwóch macierzy:

$[a, b, c, d, ..., k] * [x^(k-1)n/k, x^(k-2)n/k, ..., x^1, x^0] * [a, b, c, d, ..., k] * [x^(k-1)n/k, x^(k-2)n/k, ..., x^1, x^0]$

Macierz kx1 * 1xk = kxk

Na samej górze będą tylko elementy z a.

Potęgi będą:

$[x^(2k-2)n/k, x^(2k-3)n/k, x^(2k-4)n/k, x^(2k-5)n/k, ..., x^(k-1)n/k]$

Poniżej z b:

$[x^(2k-3)n/k, x^(2k-4)n/k, x^(2k-5)n/k, x^(2k-6)n/k, ..., x^(k-2)n/k]$

Później:

$[x^(2k-4)n/k, ...]$

Dla każdego wiersza k-1 elementów będzie taka sama jak w poprzednim.

W takim razie ile będzie unikalnych elementów?

Pierwszy element z pierwszego wiersza

ostatni element z ostatniego wiersza

i wszystkie elementy między $[x^(2k-3)n/k, x^1] -> [x^1, x^(2k-3)n/k]$

między 2 a 3 są 3 - 2 + 1 elementów

między 2k-3 i 1 jest 2k-3 - 1 + 1 = 2k-3 elementów

dodać element pierwszy z pierwszej oraz ostatni z ostatniej - 2

czyli 2k-1 elementów.

Czyli nie jest szybszy od algorytmu mnozenia, bo jego zlozonosc to bedzie $O(n^logk(2k-1))$, czyli tak jak algorytm mnozenia

4 Zadanie 4

Jeśli dzielimy wobec prostej to robimy następujący algorytm łączenia wyników:

1. Wybieramy najbardziej wysunięty na prawo punkt z lewej otoczki (p) i najbardziej wysunięty na lewo punkt z prawej otoczki (q).

2. punkt p pozostaje nieruchomo, w nim zaczepimy nasza "wskazówkę" (prostą z p do q). Wyznaczamy punkt q', który będzie kolejnym wierzchołkiem w prawej otoczce, idąc zgodnie z ruchem wskazówek zegara. Teraz sprawdźmy, jak przesunęła się nasza wskazówka, jeśli przeciwnie do ruchu wskaówek zegara to nasze q' to nowe q i powtarzamy ten krok, wpp. przechodzimy do punktu 2).

3. punkt q pozostaje nieruchomo, w nim zaczepimy nasza "wskazówkę" (prostą z q do p). Wyznaczamy punkt p', który będzie kolejnym wierzchołkiem w lewej otoczce, idąc przeciwnie z ruchem wskazówek zegara. Teraz sprawdźmy, jak przesunęła się nasza wskazówka, jeśli zgodnie z ruchem wskaówek zegara to nasze p' to nowe p

i powtarzamy ten krok, wpp. przechodzimy do punktu 3).

4. powtarzamy punkty 1) i 2), aż do momentu, w którym

i się "ustabilizują" tzn. nie będą się już zmieniały

Zauważmy, że algorytm ten zachowuje własność STOP-u, ponieważ otoczki wypunkłe, które tworzymy w każdym z kroków, są konstruowane na podstawie dwóch wielokątów wypukłych.Zatem jeżeli znajdziemy wierzchołek, dla którego wybór kolejnego wierzchołka nie jest już poprawny, to wiemy, że każdy kolejny

Analogicznie, szukamy dolnej granicy.

Porównajmy:
jak wyznaczamy kiedy ok?
górna q zgodnie przeciwnie
górna p przeciwnie zgodnie
dolna q przeciwnie zgodnie
dolna p zgodnie przeciwnie

Na podstawie Cormen strona 1040:

Aby sprawdzić, czy odcinek skierowany qp jest położony zgodnie z ruchem wskazówek zegara w stosunku do odcinka skierowanego qp' względem ich wspólnego końca q, wykonujemu przesunięcie punktu q do początku układu. To znaczy, oznaczmy jako $p - q$ to wektor $p_1' = (x_1', y_1')$, gdzie $x_1' = x_1 - x_0$, a $y_1' = y_1 - y_0$ i podobnie definiujemy $p' - q$. Następnie obliczamy iloczyn wektorowy $(p-q) \times (p'-q) = (x_1 - x_2) * (y_2 - y_0) - (x_2 - x_0) * (y_1 - y_0)$. Jeśli jego wartość jest dodatnia to odcinek skierowany qp jest położony zgodnie ze wskazówkami zegara, w stosunku do qp' (jeśli jest ujemna to przeciwnie).

Algorytm:

Przechowujemy wierzchołki znajdujące się na otoczkach L i P (lewa i prawa) na listach dwukierunkowych. Będziemy na niej przechowywać współrzędne punktu oraz wskaźnik do poprzedniego i następnego wierzchołka.

zwraca 1 jeśli zgodnie
zwraca 0 jeśli przeciwnie

```
funkcja czy_zgodnie(a, b, b')
    fi = (b.x - a.x)(b'.y - a.y) - (b'.x - a.x)(b.y - a.y);
    if fi > 0 : return 0
    wpp.: zwroc 1

funkcja krawedz_gorna():
    p = najbardziej wysuniety na prawo wierzcholek w L
    q = najbardziej wysuniety na lewo wierzcholek w P
    p' = p.poprzedni
    q' = q.nastepny
    Wykonaj:
            flaga = 0
            Dopoki czy_zgodnie(p, q, q') == 0:
                    q = q'
                    q' = q'.nastepny
                    flaga = 1
            Dopoki czy_zgodnie(q, p, p') == 1:
                    p = p'
                    p' = p'.poprzedni
                    flaga = 1
    Dopoki flaga == 1;


funkcja krawedz_dolna():
    p = najbardziej wysuniety na prawo wierzcholek w L
    q = najbardziej wysuniety na lewo wierzcholek w P
    p' = p.nastepny
    q' = q.poprzedni
    Wykonaj:
            flaga = 0
            Dopoki czy_zgodnie(p, q, q') == 1:
                    q = q'
                    q' = q'.poprzedni
```

```
                    flaga = 1
          Dopoki czy_zgodnie(q, p, p') == 0:
                    p = p'
                    p' = p'.nastepny
                    flaga = 1
Dopoki flaga == 1;

wpp.: zwroc 0
```

Definicja: Wielokąt wypukły to taki, w którym wszystkie kąty mają miary <= 180

Poprawność:

Wiemy że otoczka wypukła, którą wyzanczył powyższy algorytm zawiera wszystkie wierzchołki, ponieważ za każdym razem złączamy dwie otoczki wypukłe, a one zawierają wszystkie punkty swojego zbioru.

Wyznaczona otoczka jest wielokątem wypukłym, ponieważ wiemy, że algorytm znajduje najwyżej położoną prostą, której zmiana jednego z wierzchołków na kolejny wierzchołek poprzedniej otoczki liniowej spowodowałby wykluczenie z otoczki co najmniej jednego punktu, który w tej otoczce powinien sie znajdować.

Działania opieramy na figurze wypukłej, czyli figurze, której kąty wewnętrzne nie przekraczają 180 stopni. Jeżeli więc za punktem q', który nie spełnia warunku naszego algorytmu, znalazłby sie punkt położony wyżej od aktualnego q to okazałoby sie, że jest to figura wklęsła.

Wiemy że gdybysmy "przeszli się" po otoczce liniowej to zawsze będziemy skręcać w jedną stronę - tylko w lewo lub tylko w prawe.

5 Zadanie 5

Nasz algorytm zaczynamy w korzeniu i wygląda on tak:

1. Odkryj wierzchołek i jego dwoje dzieci.
2. Jeśli jest to minimum lokalne to zwróc wynik, wpp. 3
3. Jeśli lewe dziecko jest mniejsze od prawego to idź w lewo, wpp. idź w prawo. i przejdź do kroku 1

Ponieważ zaczynamy w korzeniu nigdy nie musimy się przejmować odkrywaniem ojca wierzchołka, ponieważ korzeń nie ma ojca, a każdy następny wierzchołek będzie wiedział skąd przyszedł czyli znał ojca.

Jeśli drzewo nie jest ukorzenione to zaczynamy w dowolnym wierzchołku i odkrywamy wszystkich sąsiadów (4 - dwoje dzieci i ojca) i idziemy do najmniejszego z nich, a później już możemy korzystać z naszego algorytmu.

Złożoność to oczywiście wysokość drzewa, czyli O(logn), a ilość operacji to 3logn jeśli drzewo ukorzenione. Jeśli drzewo nie jest ukorzenione, to możemy zacząć w liściu i iść do liścia po drugiej stronie, czyli mamy 4 * 2log(n) = 8logn operacji.

Zauważmy, że algorytm nigdy nie używa backtrackingu, czyli po prostu zawsze idzie jakąś scieżką i się kiedyś kończy (jak dojdzie do liścia)

Dlaczego znajduje minimum lokalne?

Lemat 1:

W każdym drzewie w którym wartości krawędzi są różne musi istnieć minimum lokalne.

Dowód:

W szczególności istnieje najmniejszy element w tym drzewie, a z założenia o różnych wartościach krawędzi wynika, że nie ma dwóch równych wartości, więc skoro ma najmniejszą wartość to jest minimum lokalnym.

Teza: Algorytm znajduje minimum lokalne.

Dowód:

Załóżmy nie wprost, że algorytm nie znajduje minimum lokalnego. Czyli załóżmy, że wierzchołek v który zwraca nasz algorytm nie jest minimum lokalnym. Czyli w takim razie musi istnieć jakiś wierzchołek u, który jest sąsiadem v i ma mniejszą wartość niż v. No ale jeśli istnieje taki wierzchołek, to albo

a) przyszliśmy z niego - no ale wtedy jeśli u byłby minimum lokalnym to algorytm by się skończył wcześniej

b) jest jednym z naszym dzieci - no ale wtedy algorytm by się nie skończył i poszedłby do u.

W obu przypadkach dochodzimy do sprzeczności.

W takim razie nie może istnieć taki wierzchołek u, więc v jest minimum lokalnym.

Dlaczego nie można szybciej niż logN?

Załóżmy, że istnieje algorytm który znajduje wynik szybciej niż logN. W takim razie nasz algorytm musi gdzieś zacząć, a później podjąć jakieś decyzje o wyborze kolejnych wierzchołków. W takim razie weźmy taki przykład, w którym wierzchołek jest oddalony o logN od wierzchołka w którym zaczynamy. Nawet jeśli algorytm idzie jak A* do celu z jakąś heurystyką, to odkryje co najmniej logN wierzchołków. Oczywiście jest to niemożliwe, żeby algorytm szedł od razu do poprawnego wierzchołka, ponieważ nie zna wartości wierzchołków, więc nie może podejmować żadnych decyzji w heurystyce.

6 Zadanie 6

Zadanie bazowe (podpunkt a):

1. Bierzemy dowolny wierzcholek v
2. dist = BFS(v)
3. Robimy hashmape (odleglosc od v, liczba wierzcholkow)
4. Dla kazdego poddrzewa v po kolei:
- dla kazdego wierzcholka u w poddrzewie:
dodajemy do wyniku wartosc C - dist[u] w hasmapie
- dodajemy wierzcholki z poddrzewa do hashmapy (odleglosci)
5. Usuwamy v z grafu
6. Wywolujemy sie rekurencyjnie dla powstalych poddrzew

Dlaczego obliczymy wszystko - za kazdym razem liczymy kolejno sciezki przechodzace przez rozny v (to jest warunek - musza przejsc przez v bo uwzgledniamy tylko sciezki z jednego poddrzewa do innego)

Dlaczego nie obliczymy niczego dwa razy? Bo jesli poddrzewo a ma dostep do poddrzewa b to poddrzewo b musialo byc juz wczesniej dodane, wiec nie mialo dostepu do poddrzewa a.

Zadanie zaawansowane (podpunkt b):

Szukamy centroidu drzewa (wierzchołka którego usunięcie spowoduje, że żadne z jego poddrzew nie będzie miało więcej niż połowę wierzchołków w całym drzewie).

Jak szukamy centroidu?

Bierzemy losowy wierzchołek drzewa. Jeśli spełnia warunek centroidu drzewa, to go znaleźliśmy wpp. idziemy do poddrzewa z największym rozmiarem.

W każdym drzewie istnieją co najwyżej dwa centroidy (Jordan theorem), więc możemy wybrać dowolny z nich.

Algorytm:

1. Ustalamy zmienną globalną total = 0, która mówi nam o ilości wierzchołków w poddrzewie.

2. Liczymy preprocessing - wielkości poddrzew za pomocą DFS z DP i zapisujemy wielkości poddrzew do tablicy sz[] i jednocześnie zwiększamy total o 1 w każdym wywołaniu rekurencyjnym DFS, przez co w total mamy wielkość drzewa obecnie rozpatrywanego.

3. Znajdujemy centroid drzewa:

a) Przechodzimy po wszystkich sąsiadach i idziemy do tego który ma poddrzewo większe niż tot / 2.

b) jeśli nie istnieje taki sąsiad, to nasz obecny jest centroidem i zwracamy go.

4. Odpalamy dfs który liczy ilość ścieżek o określonej odległości od centroida. (czyli dfs zaczyna z wagą 0 i dla każdego sąsiada dodaje wagę krawędzi) (oczywiście musimy skończyć iść dalej jeśli przekroczymy k, bo się nie opłaca i nie ma gdzie tego zapisać)

5. Dodajemy do wyniku ilość ścieżek które mają odległość K.

6. Dla każdego poddrzewa:

a) Odpalamy dfs który zmniejsza ilość ścieżek o określonej odległości od centroida (robi przeciwną rzecz co 4.)

b) Obliczamy podwynik - odpalamy dfs z d = 1:

- dodajemy do wyniku ilość ścieżek o odległości k-d jeśli k-d > 0

- zwiększamy d o 1 przy odwiedzeniu sąsiada

Z tego mamy ścieżki o długościach od 1 do największego poddrzewa.

7. Rekurencja dla każdego poddrzewa.

Jaka jest nasza złożoność?

Nasz dfs z punktu 2 w ogólności będzie przechodził po wszystkich wierzchołkach oprócz tych usuwanych (które były centroidami).

Mamy więc złożoność $O(n)+O(n-1)+O(n-3)+O(n-7)+O(n-15)+ ... = O(n)+O(n-2^0)+O(n-2^1)+O(n-2^2)+O(n-2^3)+O(n-2^k) = O(nlogn)$

$f(n) = O(g(n))$ gdy istnieje $c > 0, f(n) < cg(n)$

Znajdowanie centroidu drzewa - O(n)

Dlaczego centroid szukamy w O(n)?

W pierwszym kroku w najgorszej opcji znajdziemy centroid w n/2 krokach.

Później w n/4

Później w n/8

itd.

Łącznie jest to O(n)

Drzewo centroidowe ma głębokość logn, więc mamy złożoność O(nlogn).

Zbudowanie każdego poziomu drzewa centroidowego to O(n)

Wysokość drzewa centroidowego to O(logn)

Usuwanie krawędzi - n krawędzi i każde usunięcie kosztuje O(logn). Łącznie O(nlogn)

(można też nie usuwać krawędzi i zapamiętywać które usuwaliśmy przez co zwiększamy trochę pamięć)

Dfs z punktu 4 - tak samo jak ten z pkt 2.

Łączna złożonosc O(nlogn).

Dlaczego znajduje wynik poprawnie.

Dla centroida liczymy ręcznie odległości równe k.

Dla poddrzew liczymy odległości równe k - d gdy k - d > 0, czyli jeśli istnieje droga o długości k - d to możemy dołożyć do naszej obecnej odległości i będzie równa k.

W takim razie znajdujemy odpowiedzi poprawnie.

```
#include<bits/stdc++.h>
using namespace std;
using ll = long long;
const int N = 50002;
const int K = 502;
vector<int> g[N];
int n, k, sz[N], lvl[N];
int tot, done[N], cenpar[N], cnt[K];
void calc_sz(int u, int p) {
        tot++;
        sz[u] = 1;
        for (auto v : g[u]) {
                if(v == p || done[v]) continue;
```

```
                calc_sz(v, u);
                sz[u] += sz[v];
        }
}
void dfs(int u, int p, int d, int val) {
        cnt[d] += val;
        for (auto v: g[u]) {
                if (v == p || done[v]) continue;
                dfs(v, u, d + 1, val);
        }
}
long long res = 0;
void calc(int u, int p, int d) {
        if (k - d > 0) res += cnt[k - d];
        for (auto v: g[u]) {
                if (v == p || done[v]) continue;
                calc(v, u, d + 1);
        }
}
int find_cen(int u, int p) {
        for (auto v : g[u]) {
                if(v == p || done[v]) continue;
                else if(sz[v] > tot / 2) return find_cen(v, u);
        }
  return u;
}
void decompose(int u, int pre) {
        tot = 0;
        calc_sz(u, pre);
        int cen = find_cen(u, pre);
        // calculating ans
        dfs(cen, pre, 0, 1);
        res += cnt[k];
        for (auto v: g[cen]) {
                if(v == pre || done[v]) continue;
                dfs(v, cen, 1, -1);
                calc(v, cen, 1);
        }
```

```
        cenpar[cen] = pre;
        done[cen] = 1;
        for(auto v : g[cen]) {
                if(v == pre || done[v]) continue;
                decompose(v, cen);
        }
}
void solve() {
        cin >> n >> k;
        for (int i = 1; i < n; i++) {
                int u, v;
                cin >> u >> v;
                g[u].push_back(v);
                g[v].push_back(u);
        }
        decompose(1, 0);
        cout << res << '\n';
}

int32_t main() {
    ios_base::sync_with_stdio(false);cin.tie(NULL);//cout.tie(NULL);
    solve();
    return 0;
}
```

7 Zadanie 7

Merge sort działa tak, że dzielimy tablicę tak długo aż będzie miała rozmiar 1, a później scalamy mniejsze tablice.

W takim razie jeśli dojdziemy scalania tablic, to mamy:

[x] [y]

jeśli y > x to 1

później

[ab] [cd]

dla każdego elementu mniejszego od c, d dodajemy 1.

itd.

czyli w ogólności jeśli mamy do scalenia dwie tablice:

T1, T2

to ilość inwersji jest równa sumie ilości elementów mniejszych w tablicy po lewej dla każdego elementu z tablicy po prawej

MOC ZBIORU $\forall_{i \in T1} \forall_{j \in T2} i < j$

Dlaczego działa?

Wiemy, że w lewej tablicy zawsze będą elementy z mniejszym indeksem niż te w prawej, więc obojętnie w jakim porządku stoją, to jeśli element jest w lewej tablicy to był gdzieś na lewo elementu z prawej, więc mamy inwersję.

8 Zadanie 8

gówna nie tykam

9 Zadanie 9

skip