Задача A. Рухомі точки

Щоб розв'язати цю задачу, вам потрібно було прийти до наступних спостережень:

1. Завжди оптимально для кожної крапки вибрати напрямок її руху та потім рухати її в тому напрямку. Це нескладно побачити, оскільки якщо ми хочемо перемістити крапку з позиції x до $x+d$, то ми можемо витратити лише d ходів, і всі вони будуть виконані в одному напрямку.

2. Якщо крапку не можна об'єднати з іншою крапкою, то вона може не рухатися взагалі. Ми хочемо мінімізувати кількість крапок, що залишилися на лінії, тому якщо переміщення цієї крапки не впливає на кількість крапок, то ми можемо взагалі не рухати цю крапку.

3. Нехай $x_1, x_2, x_3, \ldots, x_n$ $(x_i \leqslant x_{i+1})$ є початковими позиціями крапок, а $d_i = x_{i+1} - x_i$. Ми можемо помітити, що для того, щоб об'єднати дві підрядкові крапки i та $i+1$, нам потрібно витратити щонайменше d_i часу. Доведімо, що для обраної підмножини крапок A ми можемо об'єднати всі крапки, використовуючи рівно $\sum_{j \in A} d_j$ часу. Щоб об'єднати всі крапки за цей час, ми можемо переміщати всі крапки, наприклад, вправо. Переберемо всі індекси i від 1 до $n-1$, якщо i належить до A, то ми можемо перемістити i-ту точку вправо та об'єднати з іншою крапкою.

Ми знаємо, що для заданої підмножини рухомих точок A ми можемо отримати відповідь, яка дорівнює $n - |A|$, витративши на це $\sum_{j \in A} d_j$ часу. Ми можемо мінімізувати необхідний час, вибираючи найменші можливі значення d_j. Один зі способів реалізувати це — створити масив d, відсортувати його, створити інший масив $pref_i = \sum_{j=1}^{i} d_j$ і бінарним пошуком знайти максимальний розмір A, якого ми можемо досягти.

Загальна часова складність складає $\mathcal{O}((n+q)\log n)$.

Задача B. Мех перестановки

Ми хочемо знайти кількість перестановок a, для яких $f(a) = F$ для заданого масиву F.

Нехай масив a на початку буде пустим. Ми будемо заповнювати його значеннями під час обходу масиву F. Якщо ми зараз перевіряємо позицію i масиву F ($F_0 = 0$), то ми повинні помістити в масив a $F_i - F_{i-1}$ значень. Ми повинні розмістити ці значення у перших i позиціях масиву a, оскільки в іншому випадку вони не впливатимуть на значення mex i-го префікса масиву a. Отже, ми повинні вибрати $F_i - F_{i-1}$ позицій з невикористаних позицій i-го префікса, нехай це значення буде $free_i$. Ми можемо підтримувати ці значення після розміщення елементів та зміни префікса. Отже, відповідь на задачу дорівнює добутку $A(free_i, F_i - F_{i-1})$ по всіх i.

Щоб швидко знаходити значення $A(n, k)$, нам потрібно знати значення $n!$ та $\frac{1}{(n-k)!}$. Перше значення можна попередньо обчислити за лінійний час. Обернені факторіали також можна обчислити за лінійний час. Нехай $inv_n = n!^{md-2}$, де $md = 998244353$. Тоді можна сказати, що $inv_i = inv_{i+1} \cdot (i+1)$.

Загальна складність розв'язку дорівнюватиме $\mathcal{O}(n)$.

Задача C. Нудна задача

Маємо деякі визначення з умови.

$$f(s) = \sum_{i=1}^{n} \sum_{j=i}^{n} |s_i - s_j|$$

$$cost(s) = \sum_{i=1}^{n} \sum_{j=i}^{n} f(s[i..j])$$

Можемо розкрити формулу для *cost* наступним чином:

$$cost(s) = \sum_{i=1}^{n} \sum_{j=i}^{n} f(s[i..j]) = \sum_{l=1}^{n} \sum_{r=l}^{n} \sum_{i=l}^{r} \sum_{j=i}^{r} |s_i - s_j|$$

Тепер можна помітити, що *cost* можна представити як:

$$cost(s) = \sum_{i=1}^{n} \sum_{j=i}^{n} |s_i - s_j| \cdot k_{i,j}$$

Для деяких цілих чисел $k_{i,j}$ — кількість разів, скільки $|s_i - s_j|$ зустрічається в формулі *cost*. Фактично, для фіксованих i, j можна сказати, що $k_{i,j} = i \cdot (n - j + 1)$, оскільки лише відрізки, які містять обидва ці елементи, можуть мати таку різницю, і кожен з цих сегментів містить її рівно один раз.

$$cost(s) = \sum_{i=1}^{n} \sum_{j=i}^{n} |s_i - s_j| \cdot i \cdot (n - j + 1)$$

Тепер введімо функцію $g(i)$ і представимо *cost* за допомогою g:

$$g(i) = \sum_{j=1}^{i} |s_i - s_j| \cdot j \cdot (n - i + 1) + \sum_{j=i}^{n} |s_i - s_j| \cdot i \cdot (n - j + 1)$$

$$2 \cdot cost(s) = \sum_{i=1}^{n} g(i)$$

$$cost(s) = \frac{\sum_{i=1}^{n} g(i)}{2}$$

Таким чином, якщо деякий елемент змінюється, то ми можемо оновити *cost* за час $\mathcal{O}(g)$ — час, необхідний для обчислення $g(i)$. Якщо ми оновлюємо позицію p і змінюємо її символ на c, зміна вартості дорівнює $cost_{new} = cost - g(p) + g'(p)$, де $g'(p)$ — значення $g(i)$ після зміни s_p на c.

Без будь-яких складних структур даних і обчислення g в $\mathcal{O}(n)$, ми отримуємо рішення в $\mathcal{O}(n^2 + nq)$, яке може отримати 34 бали.

Тепер ми спробуємо знайти $g(i)$ в $\mathcal{O}(\log n)$ з $\mathcal{O}(n \log n)$ попередньої обробки. Для цього ми збираємо всі дужки та модулі.

$$g(i) = \sum_{j=1}^{i} |s_i - s_j| \cdot j \cdot (n - i + 1) + \sum_{j=i}^{n} |s_i - s_j| \cdot i \cdot (n - j + 1) =$$

$$= \sum_{j=1,\, s_j < s_i}^{i} (s_i - s_j) \cdot j \cdot (n - i + 1) + \sum_{j=1,\, s_j \geqslant s_i}^{i} (s_j - s_i) \cdot j \cdot (n - i + 1) +$$

$$+ \sum_{j=i,\, s_j < s_i}^{n} (s_i - s_j) \cdot i \cdot (n - j + 1) + \sum_{j=i,\, s_j \geqslant s_i}^{n} (s_j - s_i) \cdot i \cdot (n - j + 1) \; =$$

$$= (n-i+1) \cdot (\sum_{j=1,\, s_j<s_i}^{i} (s_i \cdot j - s_j \cdot j) + \sum_{j=1,\, s_j \geqslant s_i}^{i} (s_j \cdot j - s_i \cdot j)) +$$

$$+i \cdot (\sum_{j=i,\, s_j<s_i}^{n} (s_i \cdot (n-j+1) - s_j \cdot (n-j+1)) + \sum_{j=i,\, s_j \geqslant s_i}^{n} (s_j \cdot (n-j+1) - s_i \cdot (n-j+1))) =$$

$$= (n-i+1) \cdot (\sum_{j=1,\, s_j<s_i}^{i} s_i \cdot j \; - \sum_{j=1,\, s_j<s_i}^{i} s_j \cdot j + \sum_{j=1,\, s_j \geqslant s_i}^{i} s_j \cdot j \; - \sum_{j=1,\, s_j \geqslant s_i}^{i} s_i \cdot j) +$$

$$+i \cdot (\sum_{j=i,\, s_j<s_i}^{n} s_i \cdot (n-j+1) - \sum_{j=i,\, s_j<s_i}^{n} s_j \cdot (n-j+1) + \sum_{j=i,\, s_j \geqslant s_i}^{n} s_j \cdot (n-j+1) - \sum_{j=i,\, s_j \geqslant s_i}^{n} s_i \cdot (n-j+1)) =$$

$$= (n-i+1) \cdot (s_i \cdot (\sum_{j=1,\, s_j<s_i}^{i} j - \sum_{j=1,\, s_j \geqslant s_i}^{i} j) \; - \sum_{j=1,\, s_j<s_i}^{i} s_j \cdot j + \sum_{j=1,\, s_j \geqslant s_i}^{i} s_j \cdot j) +$$

$$+i \cdot (s_i \cdot (\sum_{j=i,\, s_j<s_i}^{n} (n-j+1) - \sum_{j=i,\, s_j \geqslant s_i}^{n} (n-j+1)) \; - \sum_{j=i,\, s_j<s_i}^{n} s_j \cdot (n-j+1) + \sum_{j=i,\, s_j \geqslant s_i}^{n} s_j \cdot (n-j+1))$$

Тепер ми досягли формул сум, де кожна сума залежить лише від j. Використовуючи ці формули, ми можемо підтримувати ці значення в структурі даних, яка дозволяє додавати в точці та запитувати суму на сегменті — наприклад, дерево відрізків або бінарне індексоване дерево. Але ми все ще не знаємо, як знайти суму по всіх j, таких що $s_j < s_i$ або $s_j \geqslant s_i$. Рішення для цього досить просте, давайте підтримувати 26 таких структур даних, щоб підтримувати ці значення, i-та структура даних представляє i-ту малу літеру англійського алфавіту.

Задача D. Граф? Ви впевнені?

Спочатку ми розв'язуватимемо задачу з першим і другим типами запитів, і запити другого типу будуть задані після виконання всіх запитів першого типу. Надалі ми будемо збільшувати обмеження на c.

По-перше, нам потрібно перевірити, чи належать вершини a і b одній компоненті. Для цього ми можемо використовувати СНМ (Система Неперетинних Множин). Якщо вони належать різним компонентам, ми повинні вивести -1. Відтепер припустімо, що a і b належать одній компоненті.

1. $c = 1$. Простий шлях вважається хорошим, якщо кожне значення, записане на ребрі, має парну кількість входжень. Оскільки тут $c = 1$, всі числа, записані на ребрах, рівні, тому ми повинні перевірити, чи є довжина простого шляху парною. Довжина шляху між вершинами a і b дорівнює $d_a + d_b - 2 \cdot d_{lca(a,b)}$, де d_v позначає відстань від v до деякої кореневої вершини, а $lca(a, b)$ позначає найменшого спільного предка a і b. Оскільки нам потрібна тільки парність довжини, ми можемо сказати, що парність довжини шляху збігається з парністю $d_a + d_b - 2 \cdot d_{lca(a,b)} \equiv d_a + d_b$ ($\equiv d_a \oplus d_b$, де $\oplus$ - це операція XOR). Ми можемо вибрати будь-яку вершину як корінь, обчислити d_a і d_b з неї та використовувати для обчислення парності пізніше.

2. $c \leqslant 8$. Використовуючи ідею з попереднього рішення, ми можемо досягти алгоритму, схожого на хеш. Скажімо, що якщо колір i має непарну кількість входжень на шляху, то i-й біт числа буде дорівнювати 1. Використовуючи цю логіку, ми можемо розширити попередній алгоритм до XOR відстаней.

3. $c \leqslant 4 \cdot 10^9$. Якщо ми спробуємо застосувати той самий трюк тут, то це займе 2^c пам'яті, що очевидно занадто багато. Якщо ми почали хешувати, продовжмо хешування? Скажімо, що $hash(value) = x$, де x - випадкове число в діапазоні $[0; 2^{63})$. Тепер ми можемо перевірити, чи є шлях хорошим, дивлячись на XOR хешів значень. Якщо він дорівнює нулю, то цей шлях є хорошим, інакше ні. Чому це працює? Насправді це не працює з ймовірністю 100%. Однак ми можемо довести, що ймовірність хибно позитивної відповіді цього алгоритму настільки мала, що нею можна знехтувати.

   Як це довести? По-перше, припустімо, що всі числа, які XORуються, попарно різні. Подивимось на кожен біт окремо. Коли ми XORуємо з новим числом, ми XORуємо кожен біт з 1 або 0. Таким чином, нове значення кожного біту буде дорівнювати 0 або 1 з однаковою ймовірністю — $\frac{1}{2}$. Таким чином, мати 0 в кожному біті одночасно дорівнює $(\frac{1}{2})^b$, де b - це кількість бітів у числі.

   Оскільки у графі ми маємо $\mathcal{O}(n^2)$ шляхів, і кожен шлях може мати колізію, ймовірність колізії становить $\frac{n^2}{2^b} \approx 2 \cdot 10^{-14}$.

Тепер ми збираємося розв'язувати задачу з повними обмеженнями. Ми хочемо якось зберігати значення XOR хешів на шляху від будь-якої вершини до її кореня (оскільки може бути більше однієї компоненти). Ми можемо зробити це, використовуючи DSU та злиття від меншого до більшого. Введімо деякі визначення:

- $root$ — $root_v$ - коренева вершина вершини v;
- xr — xr_v - XOR хешів на шляху від v до $root_v$;
- sz — sz_v - розмір компоненти, з v як коренем компоненти;
- $cnt_{root,xor}$ - кількість $xr_v = xor$, таких що $root_v = root$;
- ans_v - кількість хороших шляхів у компоненті з коренем v;
- $total$ - загальна кількість хороших шляхів у всіх компонентах.

Коли ми хочемо додати ребро від a до b з номером num на ньому, ми зробимо наступне:

1. Скажімо, що $A = root_a$ і $B = root_b$;
2. Якщо $sz_A > sz_B$, тоді ми поміняємо a з b, і A з B;
3. Тепер ми додаємо всі вершини v з компоненти з коренем A, до компоненти з коренем B:
   (a) Зробимо $root_v = B$;
   (b) Зробимо $xr_v = xr_v \oplus xr_a \oplus hash(num) \oplus xr_b$. Це можна виразити наступним чином: ми йдемо від вершини до її кореня, потім до вершини a від кореня, потім проходимо нове ребро і йдемо від вершини b до її кореня. Якщо ми проходимо деякі вершини двічі, вони не будуть враховуватися в XOR;
   (c) Додамо cnt_{B,xr_v} до ans_B та $total$;
   (d) Після обробки всіх вершин додамо 1 до cnt_{B,xr_v} для кожної вершини v у компоненті з коренем A.
4. Додамо sz_A до sz_B;
5. Додамо ans_A до ans_B.

Тепер, щоб відповідати на запити, ми робитимемо наступне:

- Для запиту першого типу виконуємо описаний вище алгоритм;
- Для запиту другого типу перевіряємо, чи належать вони до однієї компоненти, $root_a = root_b$, і чи $xr_a = xr_b$. Якщо значення XOR рівні, то відповідь `YES`, в іншому випадку відповідь `NO`;
- Для запиту третього типу виводимо ans_{root_u};
- Для запиту четвертого типу виводимо $total$.

Чому це достатньо швидко? Трюк зі злиттям "від меншого до більшого"доволі поширений. Він працює за $\mathcal{O}(n \log n)$, оскільки кожен елемент зі множини змінює свій корінь не більше, ніж $\mathcal{O}(\log n)$ разів. Коли ми змінюємо кореневу вершину, це означає, що розмір множини, до якої ми додаємо ці вершини, є більшим або рівним поточному розміру. Таким чином, розмір отриманої множини принаймні вдвічі більший за розмір меншої множини.

Проте, оскільки cnt — масив map, через те, що значення XOR можуть бути занадто великими, отримуємо загальну складність задачі $\mathcal{O}(n \log^2 n)$.

Задача E. Імпульс м'ячей

Цю задачу можна вирішити за допомогою двох вказівників. Давайте відсортуємо обидва масиви так, щоб $w_i \leqslant w_{i+1}$ та $p_i \leqslant p_{i+1}$. Тепер для кожного елемента масиву w ми збираємося знайти найменший елемент у масиві p, з яким ці кульки утворять гарну пару. Пара $(i; j)$ називається гарною, якщо будь-яка інша пара $(i'; j')$ ($i \neq i'$ і $j \neq j'$) має важкість спіймати її менше або дорівнює важкості спіймати пару $(i; j)$. Очевидно, що чим більше w_i, тим менше p_j нам потрібно. Таким чином, давайте триматимемо вказівник на найменший елемент у p, який утворює гарну пару з i. Поки ми можемо утворити пару $(i; j-1)$, ми можемо зменшувати j. Отже, відповідь - це сума $n - j + 1$ для всіх значень j для кожного i.

Задача F. Типова задача на запити

Спершу, давайте проаналізуємо умови гарності відрізка, які наведені в умові:

- $len(S) \geqslant L$ — для відрізка $[l; r]$ потрібно бути $r - l + 1 \geqslant L$ або $r - L + 1 \geqslant l \Leftrightarrow l \leqslant r - L + 1$;
- $avg(S) \leqslant C$ — $\frac{\sum S_i}{|S|} \leqslant C \Leftrightarrow \sum S_i \leqslant C \cdot |S| \Leftrightarrow \sum (S_i - C) \leqslant 0$.

Таким чином, на початку ми можемо відняти C від кожного елемента a і перевірити, чи існує відрізок із сумою менше або дорівнює нулю.

Ми можемо використовувати префіксні суми для швидкого обчислення суми на відрізку, так що нехай $pref_i = \sum_{j=1}^{i} a_j$, отже $sum[l; r] = pref_r - pref_{l-1}$. Оскільки ми хочемо мінімізувати цю суму, ми можемо зафіксувати r і взяти $val = \max_{i=1}^{r-L} pref_i$, таким чином $pref_r - val$ буде мінімальною сумою відрізка S з $len(S) \geqslant L$. Якщо для будь-якого r ця сума менше або дорівнює нулю, то відповідь — `YES`.

Виконання цього призводить до складності $\mathcal{O}(n \cdot q)$.

Зробимо перше і найважливіше спостереження: якщо в якийсь момент відповідь — `YES`, то після будь-якого запиту оновлення відповідь залишиться `YES`.

Ми можемо використовувати бінарний пошук для знаходження першого моменту входження `YES` в відповіді. Давайте застосуємо перші t операцій до масиву і перевіримо, чи існує хороший відрізок S.

Як ми можемо перевірити це за лінійний час? Давайте створимо масив $b_i = a_i - a_{i-1}$ ($a_0 = 0$), отже, якщо ми додаємо x до відрізка $[l; r]$, масив b змінюється лише в двох позиціях: $b_l = b_l + x$ та $b_{r+1} = b_{r+1} - x$. Ми можемо відновити масив a так: $a_i = \sum_{j=1}^{i} b_i$.

Після знаходження першого моменту t нам потрібно вивести t `NO` відповідей і $q - t$ `YES` відповідей.

З урахуванням описаного рішення ми досягли кінцевої складності $\mathcal{O}(n \log q)$.

Задача G. Запити на дереві

Позначимо *cur* як значення `XOR` всіх вершин початкового графа, а sz_v як кількість вершин у піддереві вершини v. Простежимо, як змінюється *cur* під час виконання запитів. Якщо sz_v непарне, то значення *cur* зміниться на $cur \oplus x$, де $\oplus$ позначає операцію побітового виключного АБО, інакше воно залишиться незмінним. Пам'ятаючи про дві властивості $0 \oplus x = x$ і $x \oplus x = 0$, це спостереження є очевидним.

Тепер подивимось на *cur* по бітам. Ми можемо порахувати кількість дерев, де цей біт включений, нехай це буде $f(bit)$, тоді $ans = \sum_{bit} f(bit) \cdot 2^{bit}$. Як порахувати $f(bit)$? Розглянемо деякі випадки:

1. Жоден запит не змінює значення *bit*-го біта:
   (a) *cur* має *bit*-й біт включеним — $f(bit) = 2^q$;
   (b) Інакше — $f(bit) = 0$.
2. $f(bit) = 2^{q-1}$ інакше.

Перші два випадки досить очевидні, тепер слід довести останній випадок. Нехай cnt_0 - кількість запитів, які не змінюють *bit*-й біт у *cur*, і $cnt_1 = q - cnt_0$. Розглянемо два випадки, коли *bit*-й біт у *cur* включений і не включений.

Припустимо, що *bit*-й біт включено, тоді ми повинні вибрати парну кількість запитів, які змінюють *bit*-й біт в *cur*. Також, ми можемо використовувати будь-які запити, які не змінюють *bit*-й біт у *cur*.

$$f(bit) = 2^{cnt_0} \cdot \sum_{i=0} C(cnt_1, 2i) = 2^{cnt_0} \cdot \sum_{i=0} (C(cnt_1 - 1, 2i - 1) + C(cnt_1 - 1, 2i))$$

$$= 2^{cnt_0} \cdot \sum_{i=0} C(cnt_1 - 1, i) = 2^{cnt_0} \cdot 2^{cnt_1 - 1} = 2^{cnt_1 + cnt_0 - 1} = 2^{q-1}$$

Якщо припустити, що *bit*-й біт не включено, то можна діяти за тією ж логікою: вибрати будь-які запити, які не змінюють *bit*-й біт у *cur*, і вибрати **непарну** кількість запитів, які змінюють *bit*-й біт.

$$f(bit) = 2^{cnt_0} \cdot \sum_{i=0} C(cnt_1, 2i + 1) = 2^{cnt_0} (2^{cnt_1} - \sum_{i=0} C(cnt_1, 2i))$$

$$= 2^{cnt_0} \cdot 2^{cnt_1 - 1} = 2^{cnt_1 + cnt_0 - 1} = 2^{q-1}$$

На цьому доведення завершено. Загальна складність розв'язку дорівнює $\mathcal{O}(q \log x)$, що дорівнює $\mathcal{O}(60 \cdot q)$ у цій задачі.

Задача H. Охоронець Антон

Давайте розглянемо деякі рішення з різною асимптотикою.

Перше рішення має складність $\mathcal{O}(n!)$. Ми можемо перебрати порядок відвідування вершин, використовуючи `next_permutation`. Потім перевіряємо, чи виконується умова $deep_{order_i} \leqslant deep_{order_{i+1}}$ для всіх $1 \leqslant i < n$. Тут $deep_v$ позначає мінімальну кількість доріг, які потрібно подолати, щоб дістатися з вершини v до вершини 1. Для обчислення відстані між двома вершинами ми можемо використовувати або LCA за $\mathcal{O}(\log n)$, або просто BFS за $\mathcal{O}(n)$.

Друге рішення використовує подібну ідею. Ми можемо використовувати dp по масках, і оскільки існує лише $\mathcal{O}(2^n)$ станів, і ми легко можемо обходити їх, переходячи до вершини, до якої ми можемо дійти і де ще не були. Це працює за $\mathcal{O}(2^n \cdot n \cdot \log n)$.

Давайте зробимо перше важливе спостереження. Ми можемо подивитися на граф за рівнями. На рівні i будуть всі вершини v з $deep_v = i$. Позначимо множину цих вершин як $layer_{deep_v}$. Нехай dp_v - це мінімальна відстань для відвідування всіх вершин u, таких що $deep_u \leqslant deep_v$, дотримуючи умов в умові. Ми можемо вибрати початкову вершину на попередньому рівні та перебрати всі можливі порядки відвідування вершин на наступному рівні. Ми можемо використовувати два підходи, які вже були згадані вище. Перший алгоритм матиме складність $\mathcal{O}(\sum_{i=1}^{n} |layer_i| \cdot |layer_{i+1}|!)$, а другий - $\mathcal{O}(\sum_{i=1}^{n} |layer_i| \cdot 2^{|layer_{i+1}|})$.

Ми можемо використовувати ідею з попередніх двох підзадач - нам потрібно спробувати зробити перехід від рівня на глибині i до рівня на глибині $i+1$. Зробимо ще одне важливе спостереження:

Для множин вершин $T = layer_i$, $S = layer_{i+1}$ з $|S| > 1$, де $L = lca(S)$ (LCA - найнижчий спільний предок), нам потрібно пройти всі ребра піддерева вершини L до вершин на рівні $i+1$ один або два рази. Позначимо суму ребер в цьому піддереві як sum_edges. Конкретно, якщо ми розпочнемо з вершини u на рівні i і закінчимо на вершині v на рівні $i+1$, тоді $dp_v = dp_u + 2 \cdot sum_edges - dist(u, v)$. Ми можемо довести це наступним чином. Коли ми доходимо до будь-якої вершини p, яка має невідвідані вершини $r \in S (r \neq v)$, тоді нам потрібно перейти до цього піддерева та відвідати всі вершини r до виходу з нього. Отже, ми переходимо до сусідньої вершини та назад, проходячи кожне з цих ребер двічі. Таким чином ми відвідуємо ребра, що належать до шляху, тільки один раз.

Після того, як ми знайшли формулу для перерахунку значень dp на наступному рівні, ми досягли складності $\mathcal{O}(\sum_{i=1}^{n} |layer_i| \cdot |layer_{i+1}| \cdot \log n)$. Давайте проаналізуємо формулу $dp_v = \min dp_u + 2 * sum_edges - dist(u, v)$. Для всіх вершин $v \in S$ обчислюємо мінімальне значення $dp_u - dist(u, v)$, де $u \in T$. Це можна зробити за лінійний час. Спочатку розкладемо $dist(u, v)$ на $dist(u, LCA(u, v)) + dist(v, LCA(u, v))$, де $LCA(u, v)$ - найнижчий спільний предок вершин u і v. Оскільки ми обчислюємо мінімальне значення $dp_u - dist(u, LCA(u, v))$, ми повинні зберігати $minpath_{1,t}, minpath_{2,t}$ - два мінімальних значення $dp_u - dist(u, t); r$ у вершині t, де u знаходиться в піддереві r і $u \in T$, а r - син вершини t. Припустимо, що $minpath_{1,t,1} \neq minpath_{2,t,1}$. Для $t \in T$ ми просто зберігаємо dp_t, t, для $t \notin T$ ми обчислюємо мінімум, знаходячи два мінімальні значення $mindist_{1,r,0} - W_{u,v}; r$, де $W_{u,v}$ - вага ребра від u до v. Далі припустимо, що t дорівнює $LCA(u, v)$. Тоді:

1. $\min dp_u - dist(u, v) = \min mindist_{1,t,0} - dist(u, t)$ для $mindist_{1,t,1} \notin Parents_v$;
2. $mindist_{2,t,0} - dist(u, t)$ для $mindist_{2,t,1} \notin Parents_v$;

Де $Parents_v = parent_v \cup Parents_{parent_v}$.

Ми перерахували значення $minpath$ знизу вгору. Тепер давайте підемо в зворотньому напрямку - зверху донизу - і підтримуватимемо $value$ - мінімальне значення частини формули dp. Давайте проштовхнемо значення від v' до його дітей, тоді для всіх $u' \neq mindist_{1,v',1}$ і $parent_{u'} = v'$ ми добавимо в u' мінімальне значення $mindist_{1,v',0} - W_{v',u'}$ і $value - Wv', u'$, тоді для $u' = mindist_{1,v',1}$ ми добавимо в u' мінімум $mindist_{2,v',0} - W_{v',u'}$ і $value - Wv', u'$. В результаті ми матимемо $value$ в $u' \in S$, що позначає мінімальне значення формули переходу dp.

Таким чином, ми можемо вирішити цю задачу за $\mathcal{O}(\max deep_v \cdot n)$, що, ймовірно, буде $\mathcal{O}(n^2)$.

Тепер у нас є практично робоче рішення, давайте спробуємо подумати про деякі вершини, які насправді не впливають на відповідь. Ми можемо вилучити такі вершини. Нехай вершина v під час перерахунку dp на рівні i до рівня $i+1$ є важливою, якщо в неї є принаймні два важливих

сина. Вершини на рівні $i+1$ є важливими. Якщо вершина не є важливою, її можна вилучити з графа. Доведемо, що при наявності c вершин на рівні $i+1$ загальна кількість важливих вершин на рівнях, менших або рівних i, не перевищує c. Оскільки кожна важлива вершина має принаймні двох важливих синів, кількість вершин на попередньому рівні зменшується принаймні вдвічі. Отже, $c + \frac{c}{2} + \frac{c}{4} + \frac{c}{8} + \cdots = 2 \cdot c$.

Ми помічаємо, що ми видаляємо деякі вершини на попередньому рівні. Є два такі випадки:

- Вершина v має 0 важливих синів - ми можемо довести, що ніколи не є оптимальним розпочати з v на попередньому рівні і закінчити в якійсь вершині u на наступному рівні. Розгляньте кількість разів, коли ми проходимо через ребро від v до його батька. Оскільки ми закінчили в v, ми проходили через нього один раз на попередньому рівні. Якщо ми хочемо розпочати з вершини v, ми мусимо пройти через нього знову. Таким чином, ми пройдемо через нього двічі. Однак, якщо ми намагаємося розпочати з деякої вершини u, яка має важливих синів, кількість разів, коли ми проходимо через кожне ребро, не зміниться, за винятком ребра від v до його батька.

- Вершина v має 1 важливого сина - ця вершина вилучається, але ми використовуємо її для перерахунку значень dp.

Ми повинні бути обережні з деякими речами під час реалізації:

1. Є лише 1 вершина на наступному рівні. Нам потрібно перерахувати dp, використовуючи батька вершини;

2. Корінь дерева може бути вилучений - будьте обережні з цим.

Отже, якщо ми можемо стиснути граф за допомогою описаного алгоритму, ми можемо досягти складності $\mathcal{O}(\sum_{i=1}^{n} 2 \cdot |layer_i|)$, що дорівнює $\mathcal{O}(2 \cdot n) = \mathcal{O}(n)$.

Зрештою, ми вирішили цю задачу за лінійний час. Для кращого розуміння рекомендую вам перевірити авторське рішення.

Посилання на мій розв'язок: `https://pastebin.com/TJLQWjzC`