Задача A. Мінімальне видалення

Автор задачі: Матвій Стречень
Задачу підготував: Андрій Холод
Розбір написав: Костянтин Денисов

Блок 1

У цьому блоці $k = 10$. Оскільки всі елементи масиву лежать від 0 до 9, то після будь-яких видалень у масиві все одно не може бути числа, більшого за 9.

Тому найменше відсутнє невід'ємне ціле число завжди не більше за 10. Отже, умова вже виконана з самого початку, і відповідь дорівнює 0.

Блок 2

У цьому блоці $k = 0$. Нам потрібно, щоб mex масиву був не більшим за 0. Оскільки mex завжди є невід'ємним числом, це означає, що mex має дорівнювати саме 0.

А mex дорівнює 0 тоді і тільки тоді, коли в масиві немає числа 0. Єдина дозволена операція — видалення елемента, тому потрібно видалити всі нулі.

Отже, відповідь дорівнює кількості нулів у масиві.

Блок 3

У цьому блоці всі значення a_i попарно різні. Це означає, що кожне число від 0 до 9 зустрічається не більше одного разу.

Потрібно зробити так, щоб mex був не більшим за k. Іншими словами, після видалень має існувати хоча б одне число серед $0, 1, \ldots, k$, якого немає в масиві.

Якщо якогось числа з проміжку $0, 1, \ldots, k$ вже немає, то нічого видаляти не потрібно, і відповідь дорівнює 0.

Інакше всі числа від 0 до k присутні в масиві. Оскільки всі елементи попарно різні, кожне з них зустрічається рівно один раз. Тоді достатньо видалити будь-яке одне з цих чисел, і mex стане не більшим за k. Отже, відповідь дорівнює 1.

Блок 4

У повному розв'язку потрібно узагальнити ідею з попереднього блока. Щоб mex масиву був не більшим за k, потрібно, щоб серед чисел $0, 1, \ldots, k$ було хоча б одне відсутнє число.

Оскільки ми можемо тільки видаляти елементи, то єдиний спосіб зробити деяке число x відсутнім — видалити всі його входження з масиву. Якщо число x зустрічається cnt_x разів, то для цього потрібно рівно cnt_x операцій.

Тому достатньо порахувати кількість входжень кожної цифри від 0 до 9, а потім вибрати мінімальне значення cnt_x серед усіх x від 0 до k.

Чому це оптимально? З одного боку, якщо вибрати число з мінімальним cnt_x і видалити всі його входження, то це число стане відсутнім, а отже mex буде не більшим за k.

З іншого боку, у будь-якому коректному розв'язку після видалень має бути відсутнє хоча б одне число x з проміжку $0, 1, \ldots, k$. Для цього потрібно видалити всі його входження, тобто щонайменше cnt_x елементів. Тому менше, ніж мінімальне cnt_x, операцій зробити неможливо.

Отже, відповідь дорівнює

$$\min_{0 \leqslant x \leqslant k} cnt_x.$$

Складність розв'язку $O(n)$.

Задача B. Лінивий студент

Автор задачі: **Холод Андрій**
Задачу підготував: **Холод Андрій**
Розбір написав: **Холод Андрій**

Блок 1

Якщо $k = n \cdot m$, то операцій вистачає, щоб довільно переставити елементи в кожному рядку. Оскільки операції застосовуються до одного рядка і переставляють елементи лише в ньому, кожен рядок можна відсортувати незалежно від інших. Щоб мінімізувати суму першого стовпця, у кожному рядку ставимо найменший елемент на першу позицію; щоб мінімізувати суму другого стовпця — другий найменший, і так далі. Таким чином, достатньо відсортувати кожен рядок за зростанням і вивести суми стовпців.

Блок 2

Якщо $k = 1$ і $n = 1$, то є лише один рядок і одна операція. Нам потрібно мінімізувати суму першого стовпця, тобто мінімізувати значення $a_{1,1}$. Оскільки є лише один рядок, сума стовпця дорівнює самому елементу. Щоб зменшити $a_{1,1}$, потрібно знайти елемент рядка, менший за $a_{1,1}$, і поміняти його з $a_{1,1}$ місцями. Серед усіх менших елементів вибираємо найменший — це дасть мінімальне значення першого стовпця. Якщо такого елемента немає, операція нічого не покращує.

Блок 3

Якщо $n = 1$, то маємо один рядок і k операцій. Нам потрібно лексикографічно мінімізувати масив сум стовпців, але оскільки $n = 1$, суми стовпців — це самі елементи рядка. Обробляємо стовпці зліва направо: для кожного стовпця j знаходимо мінімальний елемент серед $a_{1,j}, a_{1,j+1}, \ldots, a_{1,m}$. Якщо він не стоїть на позиції j, витрачаємо одну операцію і ставимо його туди. Якщо $k = 0$, зупиняємось. Такий жадібний підхід коректний, бо зменшення більш лівого стовпця завжди пріоритетніше.

Блок 4

Якщо елементи у рядках початково відсортовані за спаданням, то щоб поставити на позицію j найменший елемент рядка i (який стоїть останнім), потрібно рівно одна операція: поміняти $a_{i,j}$ з $a_{i,m}$. Після цього елемент на позиції j мінімальний. Для кожного стовпця j і кожного рядка i ми можемо незалежно вирішити, чи варто витрачати операцію. Оскільки рядки відсортовані за спаданням, $b_{i,j}$ (відсортований рядок за зростанням) дорівнює $a_{i,m+1-j}$. Для кожного рядка в кожному стовпці можлива рівно одна корисна операція, тому задача зводиться до вибору найвигідніших операцій з урахуванням обмеження k.

Блок 5

Якщо $k \leqslant 10$, можна обробляти стовпці зліва направо жадібно. Для кожного стовпця j і кожного рядка i розглядаємо всі можливі операції: поміняти $a_{i,j}$ з будь-яким елементом $a_{i,j'}$, де $j' > j$ і $a_{i,j'} < a_{i,j}$. Серед усіх таких операцій вибираємо ту, що дає найбільше зменшення суми стовпця j, витрачаємо операцію і повторюємо. Оскільки $k \leqslant 10$, таких ітерацій небагато. Але треба бути обережним: після операції елемент, який ми переставили, тепер стоїть на позиції j' і може вплинути на майбутні стовпці.

Блок 6

Якщо $m = 2$, маємо лише два стовпці. Сума всіх елементів таблиці фіксована, тому $b_1 + b_2 = \text{const}$. Мінімізація b_1 автоматично максимізує b_2, але за умовою задачі нам потрібно спочатку мінімізувати b_1, а потім b_2. Оскільки $b_1 + b_2$ фіксоване, мінімізація b_1 і є єдиною метою. Для кожного рядка i можна за одну операцію поставити на першу позицію будь-який елемент цього рядка. Отже, для кожного рядка незалежно вирішуємо: якщо мінімум рядка не стоїть на першій позиції і є ще операції, витрачаємо одну операцію. Рядки обробляємо у порядку спадання виграшу від операції: спочатку ті, де різниця $a_{i,1} - \min_j a_{i,j}$ найбільша.

Блок 7

Розглянемо повне рішення. Обробляємо стовпці зліва направо. Для кожного стовпця j нам потрібно вирішити, які операції виконати, щоб мінімізувати його суму, не погіршуючи попередні стовпці.

Ключове спостереження: для рядка i і стовпця j найкраща операція — поставити на позицію j найменший елемент рядка i серед тих, що стоять на позиціях $\geqslant j$ (тобто ще не зафіксованих). Позначимо цей елемент $b_{i,j}$ (він відповідає j-му найменшому елементу рядка i, якщо рядок відсортувати за зростанням). Виграш від цієї операції дорівнює $a_{i,j} - b_{i,j}$. Якщо $a_{i,j} = b_{i,j}$, операція непотрібна.

Для кожного стовпця j збираємо всі рядки, де виграш додатній, і сортуємо їх за спаданням виграшу. Беремо перші k операцій (або менше, якщо k закінчилось). Але є тонкощі: коли ми виконуємо операцію в рядку i для стовпця j (ставимо $b_{i,j}$ на позицію j, а $a_{i,j}$ іде на позицію, де раніше стояв $b_{i,j}$), це впливає на майбутні стовпці цього рядка. Тому після операції потрібно оновити стан рядка: позиція елемента $a_{i,j}$ змінилась.

Щоб ефективно знаходити, де зараз знаходиться потрібний елемент, підтримуємо масив `pos`, де `pos`$[v]$ — поточна позиція елемента зі значенням v у своєму рядку. Після кожної операції оновлюємо `pos` для двох переміщених елементів. Для кожного рядка i заздалегідь сортуємо копію рядка b_i за зростанням — це дає нам оптимальний порядок елементів.

Таким чином, для стовпця j перевіряємо кожен рядок: якщо $b_{i,j} \neq a_{i,j}$ (тобто на позиції j стоїть не найменший доступний елемент), додаємо цей рядок до списку кандидатів із виграшем $b_{i,j} - a_{i,j}$ (від'ємним, бо $b_{i,j} < a_{i,j}$) і пріоритетом за позицією звідки беремо елемент. Сортуємо кандидатів і жадібно беремо найвигідніші операції поки є k.

Складність алгоритму — $\mathcal{O}(n \cdot m \cdot \log(n))$ через сортування кандидатів для кожного стовпця.

Задача C. Дільники

Автор задачі: **Антон Тригуб**
Задачу підготував: **Андрій Куц**
Розбір написав: **Андрій Куц**

Блоки 1–3

У перших блоках n мале, тому можна перебрати всі способи розбити числа на пари.

Після того як ми зафіксували пару чисел a та b, треба знайти мінімальну кількість операцій, щоб зробити цю пару коректною. Тобто потрібно збільшити одне або обидва числа так, щоб одне з них ділилося на інше.

У перших двох блоках значення чисел невеликі, тому це можна робити простим перебором можливих збільшень. У третьому блоці числа вже можуть бути великими, тому краще рахувати мінімальну ціну для однієї пари так само, як у блоці 8.

Після цього можна використати динаміку по підмножинах. Нехай $dp[mask]$ — мінімальна ціна, щоб розбити на пари всі числа, які входять у маску $mask$. Беремо перший ще не використаний індекс і пробуємо поставити його в пару з кожним іншим невикористаним індексом.

У кінці перевіряємо, чи мінімальна ціна не перевищує $\lfloor \frac{x}{2} \rfloor$.

Блок 4

У цьому блоці для всіх i виконується

$b_i \geqslant \lceil \frac{x}{2} \rceil$.

Посортуємо всі числа та візьмемо у пари сусідні числа. Найпростіший спосіб зробити пару коректною — зробити числа рівними. Якщо у парі стоять числа a_i та a_j, де $a_i \leqslant a_j$, то збільшимо a_i до a_j. Ціна такої пари дорівнює $a_j - a_i$.

Оскільки всі числа лежать на відрізку від $\lceil \frac{x}{2} \rceil$ до x, сумарна ціна для сусідніх пар не перевищує довжину цього відрізка, тобто не перевищує $\lfloor \frac{x}{2} \rfloor$.

Отже, таке парування завжди підходить.

Блок 5

Розіб'ємо всі числа на дві групи: числа не більші за $\lfloor \frac{x}{6} \rfloor$ та числа не менші за $x - \lfloor \frac{x}{6} \rfloor$.

Кожну групу посортуємо. Всередині кожної групи скористаємося ідеєю з четвертого блока: поставимо у пари сусідні числа та зробимо числа в кожній парі рівними.

Сумарна ціна всередині першої групи не перевищує $\lfloor \frac{x}{6} \rfloor$, бо всі числа цієї групи лежать на відрізку довжини не більше $\lfloor \frac{x}{6} \rfloor$.

Аналогічно, сумарна ціна всередині другої групи також не перевищує $\lfloor \frac{x}{6} \rfloor$.

Може статися, що після парування всередині груп залишиться по одному числу з кожної групи. Нехай це числа a_i та a_j, де $a_i < a_j$. Поставимо їх у пару. Будемо збільшувати a_j, поки воно не стане ділитися на a_i. Для цього потрібно менше ніж a_i операцій, а отже не більше ніж $\lfloor \frac{x}{6} \rfloor$.

Тому загальна ціна не перевищує

$\lfloor \frac{x}{6} \rfloor + \lfloor \frac{x}{6} \rfloor + \lfloor \frac{x}{6} \rfloor$,

а це не більше ніж $\lfloor \frac{x}{2} \rfloor$.

Блок 6

У цьому блоці відповідь можна отримати без виконання операцій, а всі числа є степенями простих чисел.

Знайдемо для кожного числа просте число та його степінь. Окремо порахуємо числа, рівні 1.

Для кожного простого числа візьмемо всі числа, які є його степенями, та розіб'ємо їх на пари. Будь-які два степені одного простого числа можна поставити в одну пару, бо менший степінь ділить більший.

Якщо кількість таких чисел непарна, то одне число залишиться без пари. Його потрібно поставити в пару з одиницею, бо 1 ділить будь-яке число. Після цього всі невикористані одиниці паруємо між собою.

Доведемо, що якщо така побудова не змогла знайти парування, то відповіді справді не існує. Єдине місце, де можуть виникнути проблеми, — це нестача одиниць для залишків від груп простих степенів. Наше парування всередині кожної групи використовує мінімально можливу кількість одиниць: якщо в групі парна кількість чисел, одиниці не потрібні, а якщо непарна — хоча б одне число все одно доведеться парувати поза цією групою. За умовою відповідь існує, тому одиниць вистачить.

Блок 7

У цьому блоці відповідь також можна отримати без виконання операцій. Кожне число є добутком не більше ніж двох простих чисел, а кожне просте число входить до розкладу всіх чисел сумарно не більше ніж двічі.

Для кожного числа знайдемо його прості дільники. Побудуємо граф, у якому вершинами будуть числа, відмінні від 1, а ребро між двома вершинами означає, що ці два числа можна поставити в одну пару. Одиниці можна парувати з будь-яким числом, тому виключимо їх з графа та будемо використовувати окремо.

Через обмеження на прості числа граф має дуже просту структуру. Жодна вершина не може мати більше двох сусідів. Інакше відповідне число мало б занадто багато простих дільників, або деякий простий дільник зустрічався б більше ніж двічі.

Розглянемо вершину, яка має двох сусідів. Єдиний можливий випадок — число в цій вершині дорівнює $a \cdot b$, а сусідні вершини відповідають числам a та b, де $a \neq b$. Будь-яка інша конфігурація порушила б умови блока.

Тому кожна компонента графа має одну з таких форм:

1. ізольована вершина;
2. пара вершин;
3. шлях з трьох вершин.

Тепер розіб'ємо всі числа на пари так, щоб використати мінімальну кількість одиниць. Ізольовані вершини паруємо з одиницями. Пари вершин паруємо між собою. У шляху з трьох вершин паруємо центральну вершину з одним із сусідів, а другого сусіда — з одиницею.

Таке парування мінімізує кількість використаних одиниць у кожній компоненті. Після цього залишок одиниць паруємо між собою. Оскільки за умовою відповідь існує, одиниць вистачить.

Блок 8

У цьому блоці гарантовано, що існує відповідь, де можна брати у пари сусідні числа початкового масиву: $(1, 2), (3, 4), \ldots, (2n-1, 2n)$.

Порахуємо мінімальну ціну, щоб зробити одну пару a, b коректною. Нехай $a \leqslant b$.

Зафіксуємо кількість операцій, які ми виконаємо з числом a. Нехай ця кількість дорівнює c_a. Тоді перше число стане рівним $a + c_a$. Тепер потрібно знайти мінімальне c_b, щоб число $b + c_b$ ділилося на $a + c_a$.

Мінімальне можливе значення $b + c_b$ — це найменше кратне числа $a + c_a$, яке не менше за b. Тому $c_b = \left\lceil \frac{b}{a+c_a} \right\rceil \cdot (a + c_a) - b$.

Отже, для фіксованого c_a ми можемо порахувати оптимальне c_b.

Залишається швидко перебрати всі суттєво різні значення c_a. Зауважимо, що кількість різних значень величини $\left\lceil \frac{b}{a+c_a} \right\rceil$ дорівнює $O(\sqrt{b})$. Це стандартний факт: для малих значень знаменника їх не більше $\sqrt{b}$, а для знаменника більшого за $\sqrt{b}$ саме значення частки вже менше за $\sqrt{b}$.

Тому можна перебрати всі різні значення цієї частки. Для кожного з них достатньо взяти мінімальне c_a, яке дає це значення. Так ми знаходимо мінімальну ціну для однієї пари за $O(\sqrt{x})$.

Повторивши це для всіх сусідніх пар, отримуємо розв'язок за $O(n\sqrt{x})$, чого достатньо для цього блока.

Блок 9: повне рішення

Посортуємо всі числа:

$a_1 \leqslant a_2 \leqslant \ldots \leqslant a_{2n}$.

Побудуємо два парування:

1. $(a_1, a_2), (a_3, a_4), \ldots, (a_{2n-1}, a_{2n})$;
2. $(a_2, a_3), (a_4, a_5), \ldots, (a_{2n-2}, a_{2n-1}), (a_1, a_{2n})$.

Для першого парування зробимо числа в кожній парі рівними. Його ціна дорівнює

$(a_2 - a_1) + (a_4 - a_3) + \ldots + (a_{2n} - a_{2n-1})$.

Для другого парування також зробимо рівними всі сусідні пари. Для останньої пари (a_1, a_{2n}) будемо збільшувати a_{2n} до найближчого кратного a_1. На це потрібно менше ніж a_1 операцій, тому ціну цієї пари можна оцінити зверху числом a_1.

Отже, ціна другого парування не перевищує

$(a_3 - a_2) + (a_5 - a_4) + \ldots + (a_{2n-1} - a_{2n-2}) + a_1$.

Уявімо пряму від 0 до a_{2n}. Доданок $a_i - a_{i-1}$ відповідає довжині відрізка між сусідніми числами, а доданок a_1 відповідає відрізку від 0 до a_1.

Відрізки, які відповідають першому паруванню, та відрізки, які відповідають другому паруванню, не перетинаються і разом лежать на відрізку від 0 до a_{2n}.

Тому сума цін цих двох парувань не перевищує a_{2n}. А оскільки $a_{2n} \leqslant x$, то хоча б одне з двох парувань має ціну не більше ніж $\lfloor \frac{x}{2} \rfloor$.

Виберемо дешевше з цих двох парувань і виконаємо описані збільшення. У кожній парі після цього одне число ділитиме інше, а загальна кількість операцій не перевищить дозволену межу.

Складність повного розв'язку становить $O(n \log n)$ через сортування.

Задача D. Розфарбуй дерево

Автор задачі: **Тарас Деркач**
Задачу підготував: **Тарас Деркач**
Розбір написав: **Тарас Деркач**

Блок 1

Оскільки $n \leqslant 10$, можна перебрати всі 2^n способів вибрати значення c_v для всіх вершин.

Для кожного розфарбування порахуємо всі значення s_v та перевіримо, чи вони попарно різні. Якщо так, додаємо 1 до відповіді.

Складність такого розв'язку — $\mathcal{O}(2^n \cdot n)$.

Блок 2

Якщо кожна вершина має не більше одного сина, то дерево є ланцюгом.

Тоді для будь-яких двох різних вершин одна з них лежить вище за іншу. Оскільки всі значення c_v додатні, то сума на шляху до нижчої вершини завжди буде більшою.

Отже, будь-яке розфарбування є коректним, тому відповідь дорівнює 2^n.

Блок 3

У цьому блоці всі вершини знаходяться на відстані не більше 2 від кореня.

Можна перебрати значення для кореня та його синів, а потім перевірити всі можливі значення для вершин на глибині 2.

Також уже тут можна помітити важливу властивість: на одній глибині не може бути більше двох вершин. Інакше для них потрібно отримати більше двох різних сум, але кожна нова вершина може збільшити суму свого батька лише на 1 або на 2.

Блок 4

У цьому блоці дерево майже є ланцюгом: не більше однієї вершини має рівно двох синів.

До розгалуження всі значення можна вибирати довільно. Якщо з'являються два сини однієї вершини, то вони мають отримати різні значення. Отже, один із них повинен отримати $+1$, а інший — $+2$.

Після цього кожна гілка знову є ланцюгом. Можна окремо перебрати вибір у місці розгалуження та перевірити, щоб суми на двох гілках не збігалися.

Втім, цей блок також покривається загальним розв'язком.

Блоки 5–8

Розіб'ємо вершини на шари за глибиною. Нехай cnt_d — кількість вершин на глибині d.

Покажемо, що якщо для деякого шару $cnt_d > 2$, то відповідь дорівнює 0.

Справді, значення кожної вершини на наступному шарі може бути лише на 1 або на 2 більшим за значення її батька. Тому з одного значення можна отримати не більше двох нових значень.

Якщо на шарі вже є дві вершини, то в коректному розфарбуванні їхні значення мають бути сусідніми. Інакше між ними залишиться проміжок, який неможливо коректно заповнити на наступних шарах. Тому наступний шар знову не може містити більше двох різних коректних значень.

Отже, якщо на будь-якій глибині більше двох вершин, коректного розфарбування не існує.

Блок 9

Тепер вважаємо, що на кожному шарі не більше двох вершин.

Корінь можна розфарбувати двома способами: $c_1 = 1$ або $c_1 = 2$. Тому спочатку відповідь дорівнює 2.

Далі розглянемо перехід між сусідніми шарами.

1. Якщо $cnt_d = 1$ та $cnt_{d+1} = 1$, то єдина вершина наступного шару може отримати або $+1$, або $+2$. Тому множимо відповідь на 2.

2. Якщо $cnt_d = 1$ та $cnt_{d+1} = 2$, то дві вершини наступного шару повинні отримати обидва значення: $+1$ та $+2$. Сам набір значень вимушений, тому на цьому переході відповідь не змінюється.

3. Якщо $cnt_d = 2$ та $cnt_{d+1} = 1$, то на поточному шарі є дві сусідні суми. Якщо нова вершина є сином вершини з меншою сумою, то один із двох варіантів заборонений, бо дає вже зайняту суму. Якщо вона є сином вершини з більшою сумою, то обидва варіанти можливі.

   Разом отримуємо 3 можливі варіанти, тому множимо відповідь на 3.

4. Якщо $cnt_d = 2$ та $cnt_{d+1} = 2$, то перехід вимушений. Новий шар знову повинен отримати дві сусідні суми, тому відповідь не змінюється.

Після всіх переходів, якщо останній шар містить дві вершини, потрібно вибрати, яка з них отримає меншу суму. Тому додатково множимо відповідь на 2.

Усі множення виконуємо за модулем $10^9 + 7$.

Загальна складність алгоритму — $\mathcal{O}(n)$ на один тестовий випадок.

Задача E. Гра на дереві

Автор задачі: **Костянтин Денисов**
Задачу підготував: **Андрій Столітній**
Розбір написав: **Едуард Тихонюк**

Загальна ідея

Зафіксуємо запит другого типу і нехай фішка спочатку стоїть у вершині s. Оскільки ходити у вже відвідану вершину не можна, гра завжди проходить уздовж простого шляху, який починається в s і закінчується у деякому листку дерева.

Підвісимо дерево за вершину s. Для вершини v позначимо через $dp[v]$ значення 1, якщо з позиції у вершині v другий гравець може гарантувати перемогу, і 0 інакше. Для листка це просто його активність:

$$dp[v] = \begin{cases} 1, & \text{якщо листок } v \text{ активний,} \\ 0, & \text{інакше.} \end{cases}$$

Позначимо через $ch(v)$ множину синів вершини v у дереві, підвішеному за s. Якщо вершина не є листком у підвішеному дереві, то відповідь залежить від того, хто робить хід з цієї вершини. На парній відстані від s хід робить перший гравець, тому він намагається отримати 0:

$$dp[v] = \min_{to \in ch(v)} dp[to].$$

На непарній відстані від s хід робить другий гравець, тому він намагається отримати 1:

$$dp[v] = \max_{to \in ch(v)} dp[to].$$

Тут to пробігає всіх синів вершини v у дереві, підвішеному за s. Такі переходи можна довести індукцією від листків до вершини s: коли значення синів уже відомі, перший гравець обирає найкращий для себе хід, а другий — найкращий для себе. Відповідь на запит дорівнює 2, якщо $dp[s] = 1$, і 1 інакше.

Нижче в блоках, де вказано $r = 1$, будемо вважати, що у всіх запитах другого типу стартова вершина дорівнює 1.

Блок 1

У першому блоці можна обробляти кожен запит незалежно.

Запит першого типу лише змінює активність одного листка. Для запиту другого типу підвішуємо дерево за задану вершину s і запускаємо DFS, який рахує описану вище динаміку. Під час DFS достатньо знати глибину від s: на парній глибині беремо мінімум по дітях, на непарній — максимум.

Один запит другого типу працює за $O(n)$, тому сумарна складність становить $O(nq)$.

Блоки 2, 3

У цих блоках стартова вершина завжди дорівнює 1, а листки тільки активуються і ніколи не стають неактивними.

Підвісимо дерево за вершину 1 і будемо підтримувати значення $dp[v]$ для всіх вершин. Спочатку всі листки неактивні, отже всі значення дорівнюють 0. Коли деякий листок активується, його dp змінюється з 0 на 1. Після цього треба пройти шляхом до кореня і перераховувати значення батьків.

Через монотонність операцій значення кожної вершини може змінитися лише з 0 на 1. Якщо під час підйому значення вершини не змінилося, то вище нічого теж не зміниться, і підйом можна зупинити.

Для кожної вершини зручно зберігати кількість синів зі значенням 1 і кількість синів зі значенням 0. Тоді:

$$dp[v] = 1$$

для вершини другого гравця тоді й лише тоді, коли хоча б один син має значення 1, а для вершини першого гравця — тоді й лише тоді, коли всі сини мають значення 1.

У третьому блоці запити зашифровані, але це не змінює структуру розв'язку: перед обробкою запиту треба лише відновити справжню вершину за формулою з умови і після запиту оновити *acc*.

Амортизована складність усіх оновлень дорівнює $O(n + q)$.

Блок 4

У цьому блоці дерево є повним бінарним. Його висота дорівнює $O(\log n)$.

Для кожної вершини зберігатимемо значення піддерева для двох можливих ситуацій: коли з цієї вершини ходить перший гравець і коли з неї ходить другий гравець. Для листка обидва значення дорівнюють його активності. Для внутрішньої вершини одне значення рахується як мінімум по дітях, а інше — як максимум по дітях.

Після зміни активності листка потрібно перерахувати тільки вершини на шляху від нього до кореня, тобто $O(\log n)$ вершин.

Щоб відповісти на запит зі стартом у вершині s, треба врахувати не лише її піддерево, а й напрямок до батька. У повному бінарному дереві шлях від s до кореня має довжину $O(\log n)$. Піднімаючись цим шляхом, можна підтримувати значення тієї частини дерева, яка лежить поза вже обробленим піддеревом: на кожному кроці додається лише брат поточної вершини та, можливо, попередньо порахована верхня частина.

Отже, і оновлення, і запит працюють за $O(\log n)$.

Блок 5

Тут лише вершина 1 може мати степінь більше ніж 2. Отже, дерево складається з кількох бамбуків, які виходять із вершини 1.

Усередині одного бамбука у кожній внутрішній вершині є рівно один можливий наступний хід, якщо не рахувати вершину, з якої ми прийшли. Тому значення вздовж бамбука просто передається від листка до центру, а тип гравця змінюється після кожного ребра.

Для кожного бамбука достатньо знати його довжину і активність кінцевого листка. Звідси можна за $O(1)$ отримати, що станеться, якщо перший хід зі стартової вершини піде в цей бік.

Якщо запит другого типу приходить у вершину 1, то перший гравець просто вибирає один з бамбуків. Тому відповідь дорівнює мінімуму значень усіх бамбуків.

Якщо ж стартова вершина лежить всередині деякого бамбука, то у першого гравця є два напрямки. Перший напрямок веде до листка цього ж бамбука, і його значення дорівнює активності цього листка. Другий напрямок веде до вершини 1. Коли фішка дійде до вершини 1, не можна йти назад у бамбук, з якого вона прийшла, тому треба об'єднувати лише всі інші бамбуки. Якщо у вершині 1 ходить перший гравець, беремо мінімум по цих бамбуках, а якщо другий — максимум. Хто саме ходить у вершині 1, визначається парністю відстані від стартової вершини до 1.

Після зміни листка змінюється тільки один бамбук. Для швидких запитів зберігаємо кількості бамбуків, які дають значення 0 та 1; коли потрібно виключити поточний бамбук, просто тимчасово не враховуємо його внесок у ці кількості. Тоді всі запити й оновлення можна обробляти за $O(\log n)$ або за $O(1)$ при акуратній реалізації з лічильниками.

Фіксований корінь і HLD

Для наступних блоків розглянемо випадок, коли всі запити другого типу мають старт у вершині 1. Підвісимо дерево за 1 і зробимо HLD-розбиття на бамбуки.

Якби між різними бамбуками не було ребер, то значення у верхній вершині бамбука просто дорівнювало б значенню листка внизу цього бамбука: уздовж бамбука в кожній вершині є лише один можливий наступний хід, тому мінімум або максимум береться з одного числа. Проблеми створюють легкі ребра в інші бамбуки.

Розглянемо вершину v поточного бамбука і легкого сина to. Якщо у v хід другого гравця, то наявність сина з $dp[to] = 1$ одразу змушує $dp[v] = 1$. Якщо у v хід першого гравця, то наявність сина з $dp[to] = 0$ одразу змушує $dp[v] = 0$.

Такі вершини назвемо позначеними. Кожна позначена вершина має своє примусове значення 0 або 1, яке вже визначене легкими синами і не залежить від продовження вниз по важкому ребру. Активний листок у нижньому кінці бамбука теж будемо вважати позначеним зі значенням 1, бо для листка dp дорівнює його активності. Для кожного HLD-бамбука будемо підтримувати set позначених вершин.

Щоб знайти значення у верхній вершині бамбука, достатньо знайти найближчу до неї позначену вершину. Якщо така вершина є, вона визначає значення всіх вершин вище в цьому бамбуку. Якщо позначеної вершини немає, то внизу бамбука лежить неактивний листок і жодне легке ребро не змушує значення змінитися, тому значення верхньої вершини дорівнює 0.

Коли змінюється значення якогось бамбука, це може змінити тільки стан його батьківської вершини у сусідньому бамбуку. Тому оновлення можна поширювати від зміненого листка вгору по дереву, переходячи між HLD-бамбуками. Кожен перехід і кожна зміна set-а коштує $O(\log n)$.

Блок 6

У шостому блоці стартова вершина всіх запитів другого типу дорівнює 1, тому достатньо описаної структури для фіксованого кореня.

Для кожного бамбука підтримуємо:

set позначених вершин, значення у верхній вершині.

Після зміни активності листка оновлюємо бамбук, у якому він лежить. Якщо значення верхньої вершини бамбука змінилося, це може змінити статус відповідної легкої дитини у батьківському бамбуку. Продовжуємо підніматися, доки значення чергового бамбука не перестане змінюватися.

Складність одного оновлення становить $O(\log^2 n)$. Відповідь на запит другого типу — це поточне значення кореня, тому такий запит працює за $O(1)$.

Блоки 7, 8

У цих блоках стартова вершина в запитах може змінюватися, тому потрібно навчитися відповідати для довільного кореня гри.

Зручно думати не про вершини, а про напрямки ребер. Нехай $val(u, p, 0)$ — результат гри, якщо фішка стоїть у u, попередня вершина дорівнює p, а хід робить перший гравець. Аналогічно $val(u, p, 1)$ — якщо хід робить другий гравець. Тоді:

$$val(u, p, 0) = \min_{to \neq p} val(to, u, 1),$$

$$val(u, p, 1) = \max_{to \neq p} val(to, u, 0).$$

Якщо доступних переходів немає, то u є листком, і обидва значення дорівнюють активності цього листка.

HLD-структура з попереднього блока підтримує саме такі значення для напрямків уздовж бамбуків. Для запиту зі стартом у s треба взяти всі напрямки з s до сусідніх вершин і застосувати мінімум, бо перший хід робить перший гравець:

$$ans(s) = \min_{to \sim s} val(to, s, 1).$$

Значення для напрямків, що лежать у легких піддеревах, беруться з відповідних бамбуків. Залишається навчитися рахувати напрямок угору, до кореня дерева 1.

Шлях від s до 1 розбивається на $O(\log n)$ HLD-бамбуків: щоразу, коли ми переходимо між двома бамбуками, ми проходимо легке ребро, а таких ребер на шляху може бути лише $O(\log n)$. Обробляємо ці бамбуки знизу вгору. Для поточного бамбука треба знайти першу позначену вершину вище поточної позиції; якщо її немає, то значення протягується до батьківського бамбука.

На межі двох бамбуків є важлива деталь. Нехай ми перейшли з попереднього бамбука в його батьківську вершину v. Під час обчислення $val(v, prev, *)$ не можна враховувати сина $prev$, бо фішка щойно прийшла з цього боку і назад ходити заборонено. Але саме цей син міг робити вершину v позначеною у структурі для фіксованого кореня. Тому для таких батьківських вершин треба окремо перевіряти позначеність після вилучення внеску попереднього бамбука. Інакше ми могли б помилково врахувати хід назад у той самий бамбук.

Для цього в кожній вершині зберігаємо кількості легких синів, які примушують значення 0 або 1. У запиті, коли входимо у батьківський бамбук через легке ребро, дивимося на ці кількості без внеску попереднього бамбука і тільки після цього вирішуємо, чи є ця вершина позначеною для поточного напрямку. Таких окремих перевірок теж $O(\log n)$, бо вони виникають лише на межах HLD-бамбуків.

Початкове HLD-розбиття і побудова структур для усіх бамбуків можна зробити за $O(n \log n)$. Після цього отримуємо $O(\log^2 n)$ на кожне оновлення і $O(\log^2 n)$ на кожен запит.

Підсумкова складність:

$$O(n \log n + q \log^2 n).$$