Розбір задачі «Двійкові маніпуляції»

Автор: Антон Ципко
Розробник: Матвій Стречень

Спершу навчимося розв'язувати `підзадачу 4` (там, де половина одиниць, половина двійок). Для цього розділимо наш масив на дві рівні половини. Розглянемо останній елемент першої половини. Якщо це двійка, то вона має стояти в другій половині, а тому ми беремо останній (n-ий) елемент і переміщуємо його *на початок*. При цьому останній елемент першого блоку переміститься на першу позицію правого блоку, тобто кількість двійок у другому блоці **не зменшиться** (може залишитись такою самою, як і до цього, за рахунок останнього елемента, який ми перемістили на першу позицію). Якщо ж так сталося, що останній елемент першого блоку — одиниця, то ми просто переміщуємо його на початок. Так продовжуємо до моменту, коли ліворуч стоять усі одиниці, а праворуч двійки. Слід зазначити, що цей процес **скінченний**, більше того - використає $O(n)$ операцій. Доведення цього факту досить тривіальне, тому залишаємо його як вправу для читача.

Тепер повернемося до основної задачі. Розберемося з випадком, коли всі числа різні. Очевидним чином ми можемо відсортувати масив із двох елементів, що стоять на позиціях 1 та 2 (якщо перший елемент більший, ніж другий, — переміщуємо другий на перше місце, інакше нічого не робимо).

Припустимо, що ми знаємо, як відсортувати масив розміру n (елементи якого стоять на індексах $1 \dots n$). Давайте навчимося сортувати масив розміру $2n$ (який розміщений також на індексах $1 \dots 2n$). Для початку визначимо, які елементи мають бути у лівій частині (тобто у межах індексів $1 - n$), а які мають розміщуватися праворуч. Позначимо «ліві» елементи як одинички, «праві» як двійки. Наприклад, елементи масиву $[1, 8, 6, 4, 7, 3, 5, 2]$ матимуть відповідно позначки $[1, 2, 2, 1, 2, 1, 2, 1]$. Тепер слід перемістити ліворуч числа, що помічені одиничками, а праворуч числа, що помічені двійками. Робити це ми навчилися за допомогою четвертої підзадачі. Все, що залишилось, — відсортувати дві половини. Для цього спершу відсортуємо ліву частину (робити це ми вміємо за припущенням), після цього n разів перемістимо числа з кінця в початок (по суті, обмінявши ліву половину з правою), знову відсортуємо ліву половину (хоча там зараз записані елементи, що стоять праворуч), після чого знову n разів перемістимо останній елемент масиву в початок, що дасть нам відсортований масив. Таким чином, ми вміємо для будь-якого n сортувати масив за кількість операцій переміщення, що оцінюється як $O(n \log n)$.

`Останній блок` (коли числа можуть бути однаковими) можна перетворити в передостанній (коли числа не повторюються). Варіацій, як це зробити, дуже багато, але найпростіший спосіб — домножити усі числа на 1000 і додати номер позиції.

Оцінимо складність. Кількість виконаних операцій переміщення: $O(n \log n)$ (виходячи з майстер-методу, англ. Master theorem). Часова складність оптимального розв'язку: $O(n \log n)$ (але час дозволяє використати значно більшу кількість операцій). Складність щодо пам'яті: $O(n)$.

Розбір задачі «Козак Вус і програмування»

Автор: Антон Ципко
Розробник: Марко Гришечкін

Будемо нумерувати біти регістру справа наліво починаючи з 0. Виходить, що перший справа біт регістру — біт під номером 0.

`Підзадача 1.` Підзадача розв'язується у 3 запити: $setNot(2, 1)$, $shiftLeft(2, 63)$, $shiftRight(2, 63)$. Можна здогадатися, що 0-ий біт регістра відповідає за парність числа. Якщо цей біт рівний нулю, то число парне, якщо рівний одиниці — непарне. Після операції $setNot(2, 1)$ у регістрі під номером 2 буде навпаки: якщо 0-ий біт рівний 1, то число у регістрі 1 було парним, а інакше — непарним. Після зсунення другого регістру спочатку на 63 позиції вліво, а потім на 63 вправо залишиться лише 0-ий біт, усі інші стануть рівні 0. Отже, у другому числі буде записано число 1, якщо перше було парним, і 0, якщо перше число було непарним.

`Підзадача 2.` Для розв'язання цієї підзадачі потрібно використати таку формулу: $a + b = (a\,xor\,b) + (a\,and\,b) \cdot 2$. Її коректність легко довести, розглядаючи числа a, b та їх суму у двійковій системі числення: якщо конкретний біт під номером x рівний 1 і в a, і в b, то до суми $a + b$ додається 2^{x+1}, а якщо лише в числі a або лише в числі b, то до суми додається 2^x. Операцію множення на двійку можна замінити зсувом уліво на одну позицію. 64 рази зробимо такі операції: $setValue(3, 2)$, $setXor(2, 1, 2)$, $setAnd(1, 1, 3)$, $shiftLeft(1, 1)$. За допомогою формули ми розуміємо, що після будь-якої кількості таких операцій сума перших двох чисел буде рівна сумі двох початкових чисел. Також за 64 операції число у регістрі 1 гарантовано перетвориться на нуль, адже ми 64 рази зсували його вліво. З цього маємо, що число, яке стоятиме на позиції 2, і буде відповіддю. Тому в кінці робимо операцію setValue(3, 2);

`Підзадача 3.` Розв'язавши підзадачу 2, ми навчилися знаходити суму будь-яких двох чисел. В окремому регістрі ми будемо зберігати відповідь — кількість одиниць. Будемо називати виділенням біта створення регістру, у якому залишився лише один цей біт, а всі інші перетворилися на 0. Для виділення біта під номером i потрібно зсунути регістр на $63 - i$ позиції вліво, а потім на 63 позицій вправо. У такий спосіб ми розглянемо окремо кожен з 64 бітів і просумуємо його з регістром-відповіддю. Ми вміємо знаходити суму двох чисел за 64 ітерації, а цього було не достатньо.

Потрібно помітити, що відповідь не може перевищувати 64, а отже, зберігається в перших 7 бітах. Тому при додаванні регістрів можна робити не 64 ітерації, а всього 7. Для меншої кількості операцій ми повинні помітити, що після додавання першого біта до відповіді використовується максимум 1 біт (можна робити всього 1 ітерацію замість 64), після додавання двох бітів використовується 2 біти (можна робити 2 ітерації), при додаванні 4 бітів використовується 3 біти (можна робити 3 ітерації) і так далі. Цих прискорень достатньо для отримання повного балу.

`Підзадача 4.` Для початку зробимо такі операції: $setXor(3, 1, 2)$, $setAnd(1, 1, 3)$, $setAnd(2, 2, 3)$. Після таких операцій задача зовсім не зміниться (більше число залишиться більшим), проте числа не будуть мати позицій з однаковими бітами. Після цього ми для обох чисел на позиціях 1 та 2 робимо таке: робимо операцію Or між числом і його копією, зсунутою на 1 позицію вправо, потім Or між новим числом і копією нового, зсунутого на 2 позиції вправо, потім зсунутого на 4 позиції вправо, потім на 8 і так до 32. Після цього задача знову не зміниться, тому що для кожного числа не виникне одиничного біта, старшого за той, що був найстаршим до операцій. Якщо після цього зробити xor 1-ого і 2-ого чисел, менше число перетвориться на 0, а більше матиме принаймні один біт, рівний 1. Тепер якщо проробити такі ж операції знову, але зсувати вліво, то менше число залишиться 0-ем, а більше число матиме 63-ий біт, рівний 1. Тому якщо після цього зсунути числа на 63 позицій вправо, більше число перетвориться на 1, а менше на 0.

`Підзадача 5.` Будемо зберігати відповідь в окремому регістрі. Зробимо 64 ітерації. Спочатку зробимо операцію or поточної відповіді і її ж зсунутої на 1 позицію вліво. Також на ітерації i ми відокремимо біт із початкового числа й поставимо його найстаршим бітом зсунувши, на $63 - i$ позицій вліво. Для кожного біта i зробимо 64 рази такі операції: результат операції and регістру біта i та поточної відповіді прооримо (зробимо операцію or) з певним іншим регістром, у якому будемо зберігати відповідь для ітерації $i + 1$, а сам біт i зсунемо на одну позицію вправо. Після цих 64 операцій потрібно проорити регістр із відповіддю на i-ій ітерації та на $i + 1$-ій. Якщо біт i був одиницею, то на ітерації $i + 1$ відповідь буде мати ще 1 одиницю, приписану зліва від уже

записаних. Після 64 ітерацій усі одиничні біти числа зсунуться вправо (те, що і потрібно в задачі, якщо переформулювати).

`Підзадача 6.` Розв'язавши підзадачу 4, ми навчилися перетворювати максимум з двох чисел на 1, а мінімум на 0. Також ми вміємо перетворювати число 1 у число $2^{64} - 1$ за допомогою зсувів на 1, 2, 4, ..., 32 позиції вліво. Якщо проендити (зробити операцію and) числа 0 та $2^{64} - 1$ і відповідні початкові числа, тоді максимальне число залишиться тим самим, а мінімальне перетвориться на 0. Прооривши їх (зробивши операцію or), отримаємо максимум. Зробивши аналогічні операції, але інвертувавши (операція setNot) 0 та $2^{64} - 1$, ми знайдемо мінімум. Тепер ми можемо отримати мінімум і максимум серед двох чисел. Тому ми можемо написати сортування бульбашкою, адже для двох чисел можемо в перше число записати мінімум, а в друге число записати максимум, а саме це і потрібно для алгоритму сортування.

Розбір задачі «Авіашляхи Потоколяндії»

Автор: Єгор Дубовік
Розробник: Ігор Баренблат

Визначимо $a \subset b$ — вираз, що означає, що a and $b = a$ (a є «підмаскою» b). Тут and — побітове I.

Розв'язуватимемо задачу покроково.

1. Визначимо D_i — m-бітова маска, де j-ий біт рівний 1, якщо місто з номером i належить списку з номером j, та 0 в протилежному випадку.

   Як порахувати D? Це домашнє завдання :)

2. Визначимо $cntD_i$ — кількість таких j, що $D_j = i$.

   Як порахувати $cntD$? Це домашнє завдання :)

3. Визначимо $intersect_{mask}$ — кількість авіашляхів, що можуть утворитися за використання будь-якого зі списків, що відповідають одиничним бітам m-бітової маски $mask$.

   Як порахувати $intersect$? $intersect_{mask} = \binom{sum_{mask \subset a} cntD_a}{2}$ (Для того щоб швидко порахувати значення, використовуйте ідею `super magic algo`(описано нижче)).

4. Визначимо $union_{mask}$ — кількість авіашляхів, що будуть утворені за використання списків, що відповідають одиничним бітам m-бітової маски $mask$ (помітимо, що це значення не залежить від порядку використання списків).

   Як порахувати $union$? Посилаючись на формулу включень-виключень, отримуємо: $union_{mask} = sum_{a \subset mask}((-1)^{bits(a)+1} \cdot intersect_a)$ (Для того щоб швидко порахувати значення, використовуйте `super magic algo`(описано нижче)). Тут $bits(m)$ — кількість одиничних бітів маски m.

5. Розв'язуватимемо задачу методом динамічного програмування. Визначимо dp_{mask} — максимальна кількість грошових одиниць, які можна заробити, використавши списки, що відповідають одиничним бітам m-бітової маски $mask$ у довільному порядку.

   Як порахувати dp? $dp_0 = 0$. $dp_{mask} = \max(dp_{mask\ xor\ 2^i} + f(union_{mask} - union_{mask\ xor\ 2^i}) \cdot r_i)$, де i — номери одиничних бітів m-бітової маски $mask$. Тут xor — побітове виключне АБО.

   Відповіддю до задачі є dp_{2^m-1}.

`/// super magic algo`

Нехай задається масив a розміру 2^m. Необхідно знайти значення масиву b, де $b_i = sum_{j \subset i} a_j$.

Здійснимо m ітерацій, де після ітерації з номером i значенням $b[x]$ буде $\sum a[y]$, таких що x and $y = y$, та починаючи з біта з номером i біти чисел x та y збігаються (біти нумеруються з 0).

Очевидно, що перед першою ітерацією $b[x] = a[x]$. На ітерації з номером i перерахуємо $b[x]$.

Якщо в числі x біт з номером i рівний 0, то $b_{new}[x] = b[x]$.

Якщо в числі x біт з номером i рівний 1, то $b_{new}[x] = b[x] + b[x\ xor\ 2^i]$.

Отже, легко реалізувати цей алгоритм так:

```
for (int i=0;i<(1<<m);i++){
    b[i]=a[i];
}
for (int i=0;i<m;i++){
    for (int mask=0;mask<(1<<m);mask++){
        if (mask&(1<<i)){
            b[mask]+=b[mask^(1<<i)];
        }
    }
}
```

`/// end of super magic algo`

Складність $O(n \cdot m + m \cdot 2^m)$ часу та $O(n + 2^m)$ пам'яті.

Розбір задачі «Козак Вус і найкраща країна»

Автор: Антон Ципко
Розробник: Марко Гришечкін

Помітимо, що країна утворює граф (міста — вершини, дороги — ребра). Для початку побудуємо мінімальне остове дерево цього графа, назвемо його MST. Правильна послідовність додавання ребер існує тоді і тільки тоді, коли сума цін усіх ребер MST менша за загальну кількість копійок у країні або рівна цій кількості. Якщо сума цін ребер більша за загальну кількість, то твердження, очевидно, виконується(ми не можемо заплатити за побудову всіх ребер MST).

Покажемо, що якщо сума менша за загальну кількість або рівна їй, то в такому MST завжди існуватиме ребро, яке можна побудувати. Якщо б це було не так, то виконувалася б нерівність $w_i > c_{v_i} + c_{u_i}$(ціна ребра більша за суму кількостей копійок у вершинах, які воно з'єднує) для кожного ребра MST. А з цих нерівностей випливає, що сума цін усіх доріг більша за загальну кількість копійок($w_i, c_{v_i}, c_{u_i} > 0$), що суперечить умові.

MST, у якому додається ребро i, ми перетворюємо на інший, де замість вершин v_i та u_i вже одна вершина з кількістю копійок, рівною $c_{v_i} + c_{u_i} - w_i$, а також до неї проведені всі ребра, інцидентні до вершин v_i та u_i. Така побудова, по суті, просто поєднує вершини сусіди у групи. При цьому різниця між сумою цін ребер MST та загальною кількістю копійок не змінилася, тому ми завжди зможемо знайти ребро, яке можна побудувати.

За допомогою цих умов досить легко написати розв'язок за $O(n^2)$: будуємо MST, а потім $n-1$ разів шукаємо ребро яке можна було б побудувати.

Для розв'язку за $O(n)$ будемо шукати потрібну послідовність ребер за допомогою пошуку у глибину. Запускаємо dfs з будь-якої вершини графу. В dfs ми для кожної вершини v перебираємо сина вершини v, запускаємо dfs від сина, а потім намагаємося побудувати ребро між v і сином. Якщо ребро не можна будувати, ми додаємо сина до певного стека і запам'ятовуємо для сина те ребро, яке його поєднує з v (батьком). Потрібно зазначити, що оскільки ребро не можна побудувати, то його побудова зменшує сумарну кількість монет групи. Після проходження всіх синів вершини v ми робимо наступне: доки остання вершина s стеку лежить у піддереві вершини v і доки ми можемо об'єднувати s та v, ми їх об'єднуємо і видаляємо s зі стека (ми пам'ятаємо ребро, що веде від s до його батька, а всі вершини на шляху між v та s уже об'єднані з v через властивості стеку). Через те, що сума цін ребер менша за загальну кількість монет або рівна їй, після dfs-обходу всі $n-1$ дороги будуть побудовані.

Розбір задачі «Труби»

Автор та розробник: Іван Фекете

`Підзадача 1:`

Обмеження цього блоку дозволяли просто рекурсивно перебрати всі можливі повороти всіх труб, після чого перевірити систему на замкнутість. Для оптимізації перебору можна використати той факт, що для кутової труби є 4 варіанти повороту, а для прямої — всього 2, але навіть без них можна було набрати повний бал за блок.

Складність: $O(4^{nm}nm)$.

`Підзадача 2:`

Можна зауважити, що для утворення замкнутої системи труб потрібні мінімум 4 кутові труби. Отже, в цьому блоці відповідь існує, якщо є рівно 4 кутові труби. Очевидно, якщо така замкнута система існує, то вона матиме форму прямокутника, і тоді задача зводиться до знаходження 4 кутів прямокутника, перевірки того, чи з'єднуються вони прямими трубами й чи є на полі якісь зайві труби, що не входять до прямокутника.

Складність: $O(nm)$.

`Підзадача 3:`

Оскільки кількість кутів не перевищує 8, то ми можемо перебрати всі можливі повороти саме для кутових деталей. Тоді для кожного стану кутових труб, ми зможемо або відновити фігуру, або сказати, що таке неможливо. Для цього потрібно кожну пряму трубу повернути так, щоб один з її кінців був приєднаним до кутової труби або до труби, для якої поворот уже відомий. Зауважимо, що якщо це робити за допомогою проходу по всьому полю, то ваш розв'язок не вкладеться в обмеження за часом. Щоб це виправити, потрібно робити пробіг тільки по тих фрагментах, де у вас знаходяться саме прямі труби, якщо відповідь існує, то їх буде $O(n+m)$ штук.

Складність: $O(4^c(n+m))$, де c — кількість кутових труб.

`Підзадача 4:`

Якщо в цьому блоці відповідь існує, то є рівно три варіанти того, що може знаходитись у рядку:

- обидва фрагменти не містять труб;
- обидва фрагменти містять прямі труби;
- обидва фрагменти містять кутові труби.

У другому випадку труби повинні бути повернуті вертикально, а в третьому вони мають бути з'єднані між собою, а ті кінці, що залишились, мають обидва бути повернуті вгору або вниз, залежно від того, чи є в попередньому рядку відкриті кінці труб, направлені донизу, чи ні. Якщо на якомусь етапі неможливо закрити відкриті кінці труб у попередньому рядку або в останньому рядку залишились відкриті кінці труб, то тоді замкнутої системи не існує, у протилежному випадку вона вже побудована.

Складність: $O(nm)$.

`Підзадача 5:`

Якщо в цьому блоці відповідь існує, то розв'язок складається з прямокутників, розташованих певним чином на полі. Отже, можна взяти розв'язок 2 блоку, адаптувавши його таким чином, щоб він розглядав одну клітинку рівно один раз, і отримаємо розв'язок цієї підзадачі з лінійною складністю.

Складність: $O(nm)$.

`Підзадача 6:`

Пройдемося по всіх рядках, після чого для кожного стовпчика переберемо всі можливі варіанти поворотів труб у ньому й оберемо такий, який не залишатиме відкритих труб (окрім, можливо, труб, направлених вниз). Якщо таких варіантів немає чи наприкінці залишились відкриті труби, то відповіді не існує, в протилежному вона побудована.

Складність: $O(4^m n)$.

`Підзадача 7:`

Цей блок призначений для розв'язків, які правильні, але написані неоптимально, це може бути розв'язок зі складністю $O(nm^2)$, $O(n^2m)$ чи якісь оригінальні розв'язки, не передбачені автором.

`Підзадача 8:`

Давайте проходитися по полю, починаючи з першого рядка, ітеруючись по кожному рядку зліва направо. Тоді якщо на кожній ітерації для чергової клітинки ми знаємо інформацію про існування відкритого кінця труби праворуч та вище від неї, то ми зможемо однозначно дізнатись поворот для цієї труби або визначити, що такого не існує. Для клітинки $(1, 1)$ ця інформація відома, отже, за мат. індукцією ця інформація буде відомою для всіх клітинок першого рядка. Отже, за індукцією, ми зможемо так зробити для всіх рядків. Додатково потрібно врахувати те, що останній рядок не має містити труб, у яких є відкриті кінці, направлені вниз.

Складність: $O(nm)$.

Розбір задачі «Козак Вус та цукерки»

Автор та розробник: Антон Тригуб

Перш за все, відсортуємо масив позицій, в яких лежать цукерки.

Якщо $n = 1$, то ми завжди можемо підібрати цю єдину цукерку, і відповідь 1.

Якщо $n = 2$, то ми можемо підібрати 2 цукерки лише тоді, коли вони не сусідні. Якщо ж цукерки сусідні, то відповідь 1. Таким чином ми вирішили `блок 1`.

Якщо $n = 3$, то відповідь — принаймні 2, адже ми завжди зможемо підібрати принаймні першу та останню цукерку. Подивимось, за якої умови ми можемо підібрати всі 3 цукерки: маємо мати $(a_2 - a_1) \vdots x$, $(a_3 - a_2) \vdots x$ для деякого $x > 1$. Таке x знайдеться тоді й лише тоді, коли $HCD(a_2 - a_1, a_3 - a_2) > 1$. В цьому випадку відповідь 3, інакше - 2. Таким чином ми вирішили `блок 2`.

Переведемо тепер задачу на математичну мову.

Для заданого масиву чисел потрібно знайти, для якої найбільшої кількості чисел з нього існує натуральне $x > 1$, при діленні на яке всі вони дають однакові остачі.

Подивимось, як можна вирішити задачу для даного x. Це можна зробити за $n \log n$, якщо замінити кожне число в масиві на його остачу за модулем x, відсортувати, і знайти найпоширенішу остачу двома вказівниками проходом по масиву.

Помітимо, що в якості x доцільно розглядати лише числа, що не перевищують максимального з a_i. Таким чином ми можемо вирішити `блок 3`, вирішивши задачу окремо для кожного x з $2 \leqslant x \leqslant 10^2$, і взявши максимум по цим значенням.

Для `блоку 4` помітимо, що ми можемо перебирати лише прості значення x (справді, якщо деякі числа дають однакові остачі при діленні на деяке складене число pq, то вони дають однакові остачі і при діленні на p, і при діленні на q). До 10^4 є порядку 10^3 простих чисел - асимптотика $O(10^3 n \log n)$ заходить.

Тепер помітимо, що при $x = 2$ відповідь гарантовано буде не меншою за $\lceil \frac{n}{2} \rceil$. Таким чином, в якості x нам потрібно перевірити лише прості p, відповідь для яких може бути більшою за $\lceil \frac{n}{2} \rceil$.

Припустимо, що для деякого $p \neq 2$ є принаймні $\frac{n}{2}$ чисел, що дають при діленні на p однакову остачу.

Для `блоку 5` тепер можемо скористатись наступною хитрістю: за такої умови має справджуватись $a_n \geqslant a_1 + p(\frac{n}{2} - 1) \geqslant p(\frac{n}{2} - 1)$. Таким чином, ми можемо вирішити задачу за $O(n \log n \pi(\frac{10^6}{n/2}))$, де $\pi(n)$ - кількість простих чисел, що не перевищують n.

Тепер перейдемо до `блоку 6`:

Припустимо, що для деякого $p \neq 2$ є принаймні $\frac{n}{2}$ чисел, що дають при діленні на p однакову остачу. Назвемо цю групу A. Виберемо з наших n чисел випадкову пару різних чисел. Імовірність того, що вони обидва потрапили в A не менша за $\frac{1}{4}$. Таким чином, пара не повністю потрапляє в A з імовірністю $\leqslant \frac{3}{4}$. Виберемо випадковим чином 50 пар. Імовірність того, що кожна не повністю потрапила в $A \leqslant (\frac{3}{4})^{50} \leqslant \frac{1}{10^6}$. Вірогідність дуже мала — можна вважати, що якась пара потрапила в A повністю.

Для кожної пари розглянемо всі прості дільники різниці чисел в парі. Об'єднаємо множини цих простих дільників по всім 50 парам. З імовірністю $\geqslant \frac{999999}{1000000}$ серед них буде шукане просте p. Тому можна перебрати лише прості дільники з цієї множини. Факторизацію можна провести за $50\sqrt{max(a_i)}$. Залишилося оцінити кількість різних простих дільників.

Якщо всі a_i не перевищують A, то кожна різниця не перевищує A, а отже добуток 50 різниць не перевищує A^50. Тепер залишилось дізнатись, скільки максимум різних простих дільників може мати число порядку A^{50}. Підрахунки на комп'ютері показують, що при $A = 10^9$ це число буде порядку 180. Отже, ми можемо вирішити задачу для кожного простого числа окремо, отримавши асимптотику $O(50\sqrt{max(a_i)} + 180 \cdot n \log n)$.

Розбір задачі «Кольорові книги»

Автор та розробник: Роман Білий

Якщо всі кольори однакові, то неможливо посортувати, крім випадку, коли всі книги зразу правильно стоять.

`Підзадача 1`. Навчимося розв'язувати задачу за $3n$ операцій. Зробимо функцію, яка міняє місцями будь-які 2 книги. Якщо вони різного кольору, то просто за 1 операцію міняємо їх місцями. Якщо однакового, то зробимо 3 операції, використовуючи ще одну книгу іншого кольору, ніж ці дві. Далі просто зліва направо виставляємо книги на свої місця.

Цей розв'язок можна оптимізувати до $2n$ операцій. Будемо виставляти книги на свої місця по черзі. Якщо нам потрібно поставити книгу на місце, а там стоїть книга такого ж кольору, то візьмемо книгу іншого кольору, яка ще не на своєму місці, і за допомогою її поставимо за 2 операції початкову на своє місце. Потрібно, щоб завжди була книга іншого кольору, яка ще не на своєму місці. Для цього оберемо правильний порядок, у якому виставляти книги на свої місця. Як варіант, будемо ставити книги на свої місця так, щоб завжди була хоча б одна книга кожного кольору не на своєму місці. Коли вже кожного кольору залишиться по одному, доставимо їх. Можливо і далі оптимізувати цей розв'язок.

`Підзадача 2`. Розіб'ємо перестановку на цикли. Нехай спочатку є k циклів. За одну операцію можливо збільшити кількість циклів максимум на 1. Нам потрібно досягти стану, коли всі елементи лежать в окремих циклах. Якщо ми міняємо 2 книги, які лежать в одному циклі, то цей цикл розіб'ється на 2. Тому будемо міняти книги так, щоб кожного разу брати 2 елементи з одного циклу і за $n - k$ операцій вийде посортувати.

`Підзадача 3`. Розіб'ємо перестановки на цикли. Якщо в циклі довжини l є хоча б 2 різні кольори, то його можна розв'язати за $l - 1$ операцію. Розв'язати означає поставити всі книги з цього циклу на свої місця. Для цього ми будемо виставляти по одному елементу на своє місце за одну операцію (це можна зробити, якщо 2 сусідні за циклом книги різного кольору). Але не можна, щоб залишилися книги тільки одного кольору, для цього потрібно зафіксувати порядок, схожий най той, що в підзадачі 1.

Тепер залишилися цикли, які складаються тільки з книг однакового кольору. Назвемо такі цикли поганими. Зауважимо, що якщо міняти місцями 2 елементи з різних циклів, то ці цикли об'єднуються. Зрозуміло, що кожен поганий цикл потрібно спочатку об'єднати з якимось іншим. Причому кількість об'єднань повинна бути мінімальна, бо кількість операцій буде рівна n -кількість_циклів+2*кількість_об'єднань. Ми можемо об'єднати 2 погані цикли різних кольорів, і тоді цей цикл стане хорошим. Тому нам вигідно розбити погані цикли на якнайбільше пар, де в кожній парі цикли різних кольорів. Нехай є c_i поганих циклів кольору і. Вигідно брати 2 максимальні c_i і з відповідних циклів утворювати пару. Цей розв'язок реалізовується за $O(n \log n)$.

Розбір задачі «Пироголяндія»

Автор та розробник: Данііл Смельський

Пироголяндія — це дерево, вершинами якого є пекарні, а ребрами є дороги. Тоді кожна вершина містить якесь невід'ємне число (значення вершини), а кожне ребро має один з двох станів: заблоковане чи розблоковане. Будемо вважати, що дерево має корінь у вершині 1.

`Блоки 1-5`:

- Запит типу 1: збережемо масив що відповідає за стан кожного ребра. При виклику запиту будемо змінювати стан ребра в цьому масиві.
- Запит типу 2: запустимо DFS із вершини p та відвідаємо всі вершини, які знаходяться в компоненті цієї вершини. При цьому не будемо переходити по заблокованих ребрах. Змінимо значення відвіданих вершин.
- Запит типу 3: можна викликати спочатку запит типу 5 до вершини p, щоб дізнатись її значення після застосування запиту типу 3. Після цього викликаємо запит типу 6, щоб змінити значення кожної вершини з компоненти вершини p на нуль. Замінюємо значення вершини p на підраховану суму.
- Запит типу 4: повертаємо значення вершини p.
- Запит типу 5: запустимо DFS із вершини p та відвідаємо всі вершини, які знаходяться в компоненті цієї вершини. При цьому не будемо переходити по заблокованих ребрах. Додамо до відповіді значення відвіданих вершин.
- Запит типу 6: запустимо DFS із вершини p та відвідаємо усі вершини які знаходяться у компоненті цієї вершини. При цьому не будемо переходити по заблокованих ребрах. Змінимо значення відвіданих вершин на нуль.
- Запит типу 7: візьмемо будь-яку вершину з ненульовим значенням. Якщо такої не існує, то очевидно, що відповідь нуль. Інакше запустимо з неї DFS, що буде повертати 1, якщо у піддереві вершини, з якої ми виходимо, є хоча б одна ненульова вершина, та 0 інакше. Тоді, якщо ми переходимо до наступної вершини через заблоковане ребро та DFS із неї повертає 1, то існує пара вершин (вершина, з якої ми запустили DFS на початку та певна вершина з цього піддерева) така, що шлях між ними проходить через це заблоковане ребро. Отже, до відповіді слід додати 1.

`Блок 6`:

- під час ініціалізації розіб'ємо дерево на компоненти, що обмежені заблокованими ребрами, та дамо кожній компоненті свій номер. Оскільки запити типу 1 відсутні, то множина вершин кожної компоненти змінюватися не буде. Також будемо зберігати значення кожної вершини не зовсім чесно: будемо зберігати її старе значення, а справжнє ми зможемо дізнатися, застосувавши певні операції.
- Для кожної компоненти будемо зберігати певні значення:
  1. integer sum — сума значень вершин компоненти.
  2. integer toAdd — значення, яке треба додати до кожної вершини цієї компоненти, щоб воно було рівним справжньому значенню вершини.
  3. integer lastVertex — остання вершина компоненти, до якої було застосовано запит типу 3, або нуль, якщо до вершин цієї компоненти ще не застосовувалися запити типу 3.
  4. clear — булева змінна, що рівна $true$, якщо до компоненти вже застосовували запит типу 3 або запит типу 6, та нуль інакше.
  5. cnt — кількість вершин у цій компоненті.

- Запит типу 2: візьмемо номер компоненти, у якій знаходиться вершина з номером p, та додамо до $toAdd$ значення w.
- Запит типу 3: можна викликати спочатку запит типу 5 до вершини p, щоб дізнатись її значення після застосування запиту типу 3. Після цього викликаємо запит типу 6, щоб змінити значення кожної вершини з компоненти вершини p на нуль. Замінюємо значення вершини p на підраховану суму. Змінимо значення $lastVertex$.
- Запит типу 4: оскільки кожна вершина зберігає своє старе значення, то треба до цього значення додати значення параметру $toAdd$ з її компоненти.
- Запит типу 5: оскільки значення вершини рівне $a_p + toAdd(p)$ (де $toAdd(p)$ рівне значенню параметра $toAdd$ компоненти, у якій знаходиться вершина з номером p), тоді суму значень вершин у компоненті можна визначити за формулою $sum + cnt \times toAdd$.
- Запит типу 6: розглянемо два випадки залежно від значення параметру $clear$.
  1. $clear = false$: зробимо всі операції чесно: пройдемося по кожній вершині, замінимо її значення на нуль. Значення sum та $toAdd$ також замінимо на нуль, а значення $clear$ на $true$.
  2. $clear = true$: у нас є лише одна вершина, нечесне значення якої відмінне від нуля, і це вершина з номером $lastVertex$. Отже, можна зробити те ж саме що у випадку $clear = false$, але замінити лише значення вершини з номером $lastVertex$.

Блок 7:

- щоб розв'язати цей блок, можна модифікувати розв'язок до попереднього блоку. З'являється лише проблема із тим, що кількість та склад компонент можуть змінюватись.
- Додамо до компонент два параметри:
  1. $allVertexes$ — вектор, що зберігає множину вершин, що знаходяться в цій компоненті.
  2. $activeVertexes$ — вектор, що зберігає множину вершин, що знаходяться в цій компоненті та мають нечесне значення відмінне від нуля.
- Через запит типу 1 треба вміти поєднувати певні компоненти. Для цього застосуємо структуру даних $DisjointSetUnion$ (система неперетинних множин).
- Розглянемо операцію $unionSets$, застосовану до компонент із номерами x та y у нашій DSU. Вона повинна поєднати дві компоненти. Будемо вважати, що це лідери відповідних компонент у DSU та вони різні. Ми будемо підвішувати компоненту з номером y до компоненти з номером x. Тоді нам треба якось поєднати їх параметри. Для цього спочатку очистимо список $activeVertexes(y)$, а потім пройдемося по вершинах компоненти y (список $allVertexes(y)$), якщо значення $clear = true$, тоді спочатку замінимо значення вершини на нуль, а потім замінимо значення кожної з них на $a_p = a_p + toAdd(y) - toAdd(x)$, додамо кожну з них до списку $activeVertexes(y)$, а до $sum(y)$ додамо $cnt(y) \times toAdd(y)$. Далі перекинемо вершини зі списку $allVertexes(y)$ до списку $allVertexes(x)$, а також вершини зі списку $activeVertexes(y)$ до списку $activeVertexes(x)$. Додамо до $sum(x)$ значення $sum(y)$, а до $cnt(x)$ значення $cnt(y)$. Підвісимо вершину y до вершини x у DSU.
- Які зміни відбулися у запитах?
  - Запит типу 2: змін немає, лише треба брати номер компоненти з DSU.
  - Запит типу 3: після виклику запиту типу 6 наш список $activeVertexes$ повинен містити лише вершини p, але він є порожнім, тому треба додати до нього вершину з номером p, якщо її значення після цього запиту є ненульовим.
  - Запити типу 4 та 5: відповіді можна знайти так само як у блоці 6.

  - Запит типу 6: у розв'язку до шостого блоку ми фактично проходилися по списку $allVertexes$, але помітимо, що можна проходитись лише по списку $activeVertexes$ та замінювати лише їх значення на нуль. Після цього слід видалити всі вершини зі списку $activeVertexes$.

- Асимптотика такого розв'язку не буде вкладатися в обмеження, бо може перевищувати $O(n^2)$. Проте можна застосувати техніку *small-to-large* (від меншого до більшого). Перш ніж обробляти запит типу 1, давайте упорядкуємо x та y так, щоб $cnt(x)$ було не меншим за $cnt(y)$. Тоді асимптотика такого рішення вже буде $O(n \log n)$, що повинно проходити цей блок.

Блок 8:

- оскільки компоненти не будуть змінюватись у складі, можна побудувати додаткове дерево. Для цього давайте умовно видалимо з нашого дерева всі заблоковані ребра та стиснемо кожну компоненту, що утворилася в одну вершину в новому дереві. Усі нові вершини ми якось пронумеруємо, а також для кожної вершини початкового дерева запам'ятаємо номер вершини, до якої вона увійшла в новому дереві. Тепер додамо до цього дерева заблоковані ребра, що ми видалили на початку його побудови. Підвісимо це дерево за будь-яку вершину. Будемо вважати, що вершина в новому дереві має чорний колір, якщо є хоча б одна вершина з ненульовим значенням у початковому дереві, що належить цій вершині в новому дереві. Переформулюємо запит сьомого типу: треба знайти мінімальну кількість ребер у новому дереві, які потрібно розблокувати, щоб на шляху між кожною парою чорних вершин усі ребра були розблоковані.

- Розглянемо будь-яке ребро з нового дерева. За якої умови його треба розблокувати? Якщо у піддереві нижньої вершини (більш віддаленої від кореня) цього ребра є хоча б одна вершина чорного кольору та ззовні піддерева верхньої вершини є хоча б одна вершина чорного кольору, то це ребро треба розблокувати. Інакше не існує пари вершин чорного кольору, на шляху між якими лежить це ребро, отже, його розблоковувати не потрібно.

- Запустимо DFS з кореня нового дерева та випишемо вершини чорного кольору у порядку часу входження до них. Нехай це послідовність g довжини k. Уведемо нову функцію $dist(x, y)$ — кількість ребер на шляху між парою вершин x та y у новому дереві. Звідси отримуємо формулу для визначення відповіді на запит сьомого типу: $ans = (dist(g_1, g_2) + dist(g_2, g_3) + \ldots + dist(g_{k-1}, g_k) + dist(g_k, g_1))\ /\ 2$. Чому це правильно? Знову розглянемо будь-яке ребро з нового дерева. Якщо існує вершина чорного кольору у піддереві нижньої вершини цього ребра та вершина чорного кольору ззовні верхньої вершини цього ребра, то це ребро увійде до суми рівно два рази, бо воно увійде в це піддерево та вийде з нього. Інакше воно жодного разу не увійде до цієї суми.

- Отже, для розв'язання сьомого типу запитів треба вміти підтримувати цю суму. Оскільки відсутні запити першого типу, склад нового дерева змінюватися не буде, але вершини можуть змінювати колір. Давайте зберігати таку суму: $query_7 = dist(g_1, g_2)+dist(g_2, g_3)+\ldots+dist(g_{k-1}, g_k)$ Тоді нас цікавлять лише зміни, що відбуваються у послідовності g. У нас може або додаватися новий елемент до послідовності, можливо, десь у середину, або видалятися.

- Випишемо в окремий масив усі вершини нового дерева у порядку часу входження до них при виклику DFS з кореня. Тобто коли DFS входить у нову вершину, поточний час збільшується на 1. Звідси $tin(v)$ — час входження до вершини v, а $tout(v)$ — час виходу з вершини v. Особливість цього порядку в тому, що всі вершини з піддерева вершини v зустрічаються на відрізку $tin(v) \ldots tout(v)$ у цій послідовності. Для кожної вершини запам'ятаємо позицію, на якій вона зустрічається у цьому масиві. Якщо вершина має чорний колір, то в додатковому масиві на її позицію поставимо одиничку, інакше нуль. Як треба обробляти зміни у новому дереві?

  - Зміна кольору вершини з чорного на білий. Припустимо, що ця вершина зустрічалась на позиції u в послідовності g. Тоді у формулу для знаходження значення $query_7$ вона могла увійти максимум у двох доданках.
    1. $u > 1$: $dist(g_{u-1}, g_u)$
    2. $u < k$: $dist(g_u, g_{u+1})$

Тоді нам треба перевірити можливі випадки та вміти знаходити відстань між двома вершинами, а потім віднімати їх від суми. Для цього можна скористатися додатковими структурами даних.

– Зміна кольору вершини з білого на чорний аналогічна, але тепер треба додавати числа до нашої суми.

- щоб дізнатися відповідь на запит сьомого типу, до нашої суми треба додати значення $dist(g_1, g_k)$, а потім поділити суму на два. Щоб знайти першу та останню чорну вершини, звернемося до нашого масиву з нуликів та одиничок. Можемо на цьому масиві побудувати дерево відрізків та швидко знаходити наші вершини (можна зробити *set*, але дерево відрізків нам знадобиться для вирішення наступних блоків).

Блок 9:

- Випишемо всі вершини нашого дерева у порядку часу входження в них у DFS, викликаного з кореня дерева. Тепер для кожної вершини у додатковому масиві, на позиціях де вони зустрічаються у масиві часу входження, будемо зберігати кількість заблокованих ребер на шляху від кореня до цієї вершини. Тоді всі вершини з піддерева вершини v, що ще й знаходяться в компоненті вершини v, мають те ж саме значення у додатковому масиві, що й значення вершини v. Згадаємо, що вершини піддерева вершини v знаходяться на відрізку $tin(v) \dots tout(v)$ у масиві обходу DFS. Важливий факт, який треба помітити, — значення вершини v на цьому відрізку в додатковому масиві мінімальне. А отже, усі вершини, що мають мінімальне значення на цьому відрізку, знаходяться в компоненті вершини v. Тобто якщо визначити в кожній компоненті її корінь (найближчу вершину до кореня всього дерева), тоді ця компонента може бути визначена як множина вершин, що лежать на відрізку $tin(v) \dots tout(v)$, та мають на ньому мінімальне значення.

- Побудуємо на додатковому масиві дерево відрізків. Кожен лист цього дерева буде відповідати за певну вершинку початкового дерева, визначену за номером вершини у масиві обходу DFS. У кожному листі дерева відрізків будемо зберігати пару параметрів:

  1. значення вершини
  2. значення вершини у додатковому масиві (кількість заблокованих ребер на шляху від кореня дерева до цієї вершини)

  А для інших вершинок дерева відрізків, що відповідають за певні проміжки, будемо зберігати наступні параметри:

  1. сума значень усіх вершин з цього проміжку
  2. мінімальне значення у додатковому масиві усіх вершин з цього проміжку
  3. кількість вершин із мінімальним значенням на цьому проміжку
  4. додаткові параметри, які умовно будемо називати обіцянками, щоб оброблювати певні операції на цьому дереві.

  Перейдемо до розв'язку запитів:

  – Запити типу 1: подивимося на піддерево нижньої вершини цього ребра. Оскільки це ребро заблоковане, то воно додає 1 до значення кожної вершини у додатковому масиві з цього проміжку. Тобто у дереві відрізків треба відняти 1 від додаткового значення кожної вершини з цього проміжку, а це можна зробити використовуючи обіцянку $add1$ — число, на яке треба змінити додаткове значення кожної вершини з цього проміжку.

  – Запити типу 2: спочатку знайдемо корінь компоненти, у якій знаходиться ця вершина (як це зробити, дивіться після розв'язку блоку 9). Нехай це вершина v. Тоді до значення всіх вершин з компоненти вершини v треба додати число w. Оскільки ці вершини знаходяться на відрізку $tin(v) \dots tout(v)$ та мають мінімальне значення на ньому, то можна використати наше дерево відрізків, а саме ще один параметр для обіцянок $add2$ — число, на яке треба змінити значення кожної вершини з цього проміжку.

- Запити типу 3: викличемо запит типу 5 та 6. Запит типу 6 лише замінив значення кожної вершини нашої компоненти на нуль, але значення додаткового параметру не змінилось. Отже, ми можемо додати до значення нашої вершини p суму, підраховану в запиті типу 5, використавши наше дерево відрізків для проміжку $tin(p) \dots tin(p)$.
- Запити типу 4: використовуючи обіцянки, ми можемо дізнатися значення окремої вершинки, поступово спускаючись до неї по дереву відрізків та перекидаючи обіцянки до синів поточної вершини.
- Запити типу 5: щоб дізнатися відповідь на цей запит, треба взяти суму значень усіх мінімумів на відрізку $tin(v) \dots tout(v)$, де v — корінь компоненти вершини p. Наше дерево відрізків уже зберігає цю суму для часткових проміжків, тому треба обрати головні з них та комбінувати.
- Запити типу 6: будемо використовувати ще один параметр обіцянки в дереві відрізків $clear$ — чи було очищення кожної вершинки із мінімальним значенням на проміжку цієї вершинки дерева відрізків.
- Слід зауважити, що на відміну від перекидування обіцянок у звичайному дереві відрізків (наприклад, звичайне додавання на відрізку), у цьому дереві відрізків ми будемо передавати обіцянку лише до синів із мінімальним значенням. Це стосується всіх обіцянок, окрім $add1$, яку слід передавати так само, як у звичайному дереві відрізків.
- Отже, асимптотика такого рішення буде $O(n \log n)$.

Як ефективно знаходити корінь кожної компоненти (із можливістю застосовувати запити першого типу):

- знову випишемо всі вершинки у порядку обходу DFS, а також повернемося до параметрів $tin(v)$ та $tout(v)$. Заблокованими будемо називати ті вершинки, ребро з яких до предка у дереві заблоковане. Отже, зробимо ще один додатковий масив. Якщо вершина заблокована, або це вершинка 1(корінь дерева), то на її позиції будемо зберігати значення $tout(v)$, а інакше будемо на її позиції зберігати нуль.

- Як знаходити корінь компоненти: по-перше, усі предки нашої вершини будуть знаходитись на позиціях із номерами, меншими ніж $tin(p)$, тобто на відрізку $1 \dots tin(p)$. А також, за властивістю tin та $tout$, значення $tout(v)$ повинно бути більшим або рівним за $tout(p)$, якщо v предок p. Якщо є дві вершини, що є кандидатами на корінь нашої компоненти, то серед них треба обрати останню. Тобто лідер компоненти вершини p це остання вершинка на відрізку $1 \dots tin(p)$, на позиції якої стоїть значення, що не менше за $tout(p)$.

`Блок 11`:

- Єдине, що ми ще не розглянули, це як одночасно підтримувати запити першого та сьомого типу. Тепер у нас може змінюватися структура нашого нового дерева, що було побудоване на стиснутих компонентах, тому будувати нове дерево може виявитися занадто незручно. Зробимо копію нашого початкового дерева. І тепер чорними будуть вершини, що є коренями компонент, що містять хоча б одну вершину з ненульовим значенням. Тоді змінилась функція $dist(x, y)$ — кількість заблокованих ребер на шляху між вершинами x та y. Проте ми можемо дізнатися її значення, використовуючи значення наших додаткових параметрів у дереві відрізків, що ми застосовували для вирішення перших 6 типів запитів.

- Тепер при виклику запиту типу 1 треба вміти поєднувати дві компоненти та ділити одну компоненту на дві. Це ми можемо зробити певною послідовністю фарбувань вершин з білого кольору на чорний та навпаки, залежно від складу компонент.

- Проте нам треба зауважити, що це ребро могло входити до значення $query_7$. Давайте спочатку змінимо його стан, перерахувавши значення $query_7$, а потім будемо поєднувати або ділити компоненти.

  1. Зміна стану ребра з розблокованого до заблокованого: треба дізнатися, скільки разів це ребро буде входити до суми. Згадаємо про масив, де ми зберігаємо 1 на позиціях чорних

вершин та 0 на позиціях білих. Тоді, якщо на відрізку $tin(p) \dots tout(p)$ зустрічається хоча б одна одиничка та на відрізку $1 \dots tin(p)-1$ також зустрічається хоча б одна одиничка, тоді існує доданок $dist(q, p)$, що має входити до значення $query_7$. Тобто до цього значення слід додати 1. Так само якщо на відрізку $tin(p) \dots tout(p)$ трапляється хоча б одна одиничка та на відрізку $tout(p) + 1 \dots n$ також трапляється хоча б одна одиничка, то є доданок $dist(p, q)$, що має входити до значення $query_7$, тобто це значення треба знову збільшити на 1.

2. Зміна стану ребра із заблокованого на розблоковане: аналогічно до першого випадку, проте треба віднімати 1 від значення $query_7$.

Асимптотика рішення $O(n \log n)$.