Задача 1A. Точки на прямій

Автор задачі: **Матвій Асландуков**
Задачу підготував: **Костянтин Денисов**
Розбір написав: **Костянтин Денисов**

Блок 1

Для вирішення цього блоку достатньо перебрати всі можливі пари i, j та за n запитів перевірити чи всі точки знаходяться між цією парою. Тоді матимемо рішення за n^3 запитів, що з великим запасом проходить цей блок.

Блок 2

Нехай ми знаємо лише одне число x з шуканої пари точок x, y, тобто одну з крайніх точок. Знайдемо y. Будемо поступово розглядати точки від точок з меншими індексами до більших і при цьому підтримуватимемо точку v, що відрізок з точок x, v містить всі точки, що оброблені на даний момент. Початково покладемо $v := x$. Далі коли приходить нова точка i, то перевіряємо запитом, чи точка i знаходиться між точками x та v. Якщо не знаходиться, то присвоїмо $v := i$, оскільки пара точок v, x вже не містить нової точки i. В іншому випадку v змінювати не треба. Таким чином після оброблення всіх точок, то між парами точок x, v будуть знаходитися всі інші і таким чином ми знайшли $y = v$.

Оскільки ми не знаємо x, то будемо його перебирати. Для кожного x запустимо алгоритм знаходження y (припускаючи, що x — крайня точка) та додатково за n запитів перевіримо, чи дійсно між парою точок x, y знаходяться всі інші точки (перевірка припущення). Тоді маємо рішення, що виконує $2n^2$ запитів, що вкладається в заданий ліміт.

Блок 3

Поступово розглядатимемо наші точки й підтримуватимемо пару точок x, y, що всі розглянуті точки знаходяться між ними. Нехай розглядаємо точку i, тоді запитом дізнаємося, чи ця точка знаходиться між поточною парою точок x, y. Якщо знаходиться, то очевидно, що оновлювати пару x, y з надходженням цієї нової точки не треба. Якщо ж не знаходиться, то дізнаємося запитом, чи точка y знаходиться між парою точок x, i. Якщо знаходиться, то між парою точок x, i знаходяться всі розглянуті точки, інакше між парою точок y, i. Коли розглядаємо нову точку, то в гіршому випадку витрачаємо 2 запити, щоб оновити поточну пару точок, що містить всі інші. Тоді маємо рішення, що використовує не більше $2 \cdot n$ запитів.

Блок 6

Рішення на повний бал є покращенням рішення з блоку 3, тому варто спершу прочитати його.

Нехай m — кількість точок, на яких ми витратили 2 запити, щоб оновити поточну пару x, y. Іншими словами, це такі точки, що поточний максимум або мінімум координат розглянутих точок змінився. Було б добре, якби таких точок було небагато.

Давайте спробуємо переглядати наші точки в рандомному порядку, а не в фіксованому. Виявляється, що очікувана кількість оновлень префіксних максимумів у рандомному масиві буде $O(\log(n))$. Аналогічно маємо з мінімумами. Тоді якщо переглядати наші точки в рандомному порядку, то очікуване значення m буде $O(\log(n))$. Тоді матимемо рішення, що очікувано використовує $n + O(\log(n))$ запитів, що на практиці з запасом понад у 2000 запитів вкладається в заданий ліміт.

Задача 1B. Подарунок Леді

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

Блок 1

У першому блоці $n \leqslant 5$ та $q = 0$, а це означає дві речі:

1. Через те, що нам відомі усі рядки, ми знаємо символ у кожному вузлі. Символ у вузлі під номером i дорівнює першому символу в рядку i.

2. У кожного вузла є n варіантів, куди може вести зв'язок з цього вузла. Тому сумарна кількість мереж, не враховуючи символи в вузлах, дорівнює n^n.

Маючи ці два факти, ми можемо написати просте рішення.

Спочатку знаходимо символи у кожному вузлі. Потім перебираємо усі варіанти зв'язків. Це можна зробити рекурсією або будь-яким іншим методом.

Потім для кожного варіанта перевіряємо, чи згенерована таблиця збігається з тою, що у вхідних даних. Якщо так, виводимо символи та зв'язки цієї мережі та закінчуємо виконання. Якщо ні, продовжуємо пошук.

Асимптотика рішення: $O(n^{n+2})$.

Блок 2

У другому блоці сказано, що мережа є набором пар та одиничних точок. Розглянемо ці два варіанти окремо.

Як буде виглядати рядок a_i, якщо $x_i = i$? Ми починаємо з вузла під номером i та рухаємося по зв'язках в мережі. Але зв'язок з i веде назад в i! Тому весь рядок a_i буде складатися з символу, записаного на вузлі i – s_i.

Тепер, як буде виглядати рядок для пари a, b, такого, що $x_a = b$ і $x_b = a$? Розглянемо рядок a. Перший символ дорівнює s_a, другий символ s_b (тому що ми перейшли з вузла a в вузол b через зв'язок з вузла a), третій – s_a і так далі.

Ми застрягаємо у циклі між вершиною a та b, тому рядок a дорівнює $s_a, s_b, s_a \ldots$, а рядок b – $s_b, s_a, s_b, \ldots$. Як тепер відновити початкову мережу?

Якщо ми маємо рядок з символів $abababa\ldots$, ми маємо цикл між двома вузлами, тому десь має бути рядок $bababab\ldots$. Ми можемо знайти усі такі пари та з'єднати їх між собою.

У випадку, коли рядок складається з одного символу ми кажемо, що вузол з'єднаний сам з собою, тобто $x_i = i$. Зауважте, що можлива ситуація, коли в початковій мережі дві вершини були у парі та мали однакові символи. Це означає, що їхні рядки складаються з такого самого символу. Якщо ми з'єднаємо їх самих з собою, мережа буде відрізнятись, але згенерована таблиця — ні, тому це не проблема.

Асимптотика рішення: $O(n^3)$.

Блок 3

У третьому блоці мережа є набором зірок. У такому випадку, у кожному рядку усі символи однакові, крім першого, який може відрізнятись.

Якщо у нас є рядок i типу $abbb\ldots$, вузол i має вести в інший вузол, чий рядок дорівнює $bbbb\ldots$. Тоді знайдемо для кожного символу c номер рядка $root_c$ такого, що він повністю складається з символу c. Якщо такого рядка не існує, значення $root_c$ для цього символу нас не цікавить.

Тепер для кожного вузла i поставимо $x_i = root_c$, де c – другий символ рядка i. Відомо, що рядок $root_c$ повністю складається з символу c, тому згенерована таблиця буде абсолютно ідентичною до тої, що у вхідних даних.

Асимптотика рішення: $O(n^2)$.

Блок 4

У четвертому блоці мережа є деревом з коренем у вузлі 1. Для кожної вершини v знайдемо вершину p таку, що $a_{v,[1,3n-1]} = a_{p,[0,3n-2]}$, де $a_{v,[l,r]}$ — символи рядка v з l-го по r-й включно (символи в рядку пронумеровані від 0 до $3n-1$). Цієї умови достатньо, щоб поставити $x_v = p$.

Довести це досить легко, можемо зауважити, що "шлях"кожної вершини закінчується в циклі з однієї вершини 1, тому що мережа це дерево (не існує циклів крім кореня) і $x_1 = 1$. Тоді ми можемо подивитися на рядок a_v і побачити, що в наступної вершини x_v рядка a_{x_v} повинна збігатися з рядком a_v, починаючи з другого елементу.

Знайти таку вершину можна перебором, для кожної вершини є всього є $O(n)$ кандидатів, перевірка двох рядків на рівність виконуємо за $O(n)$.

Асимптотика рішення: $O(n^3)$.

Блок 5

Мережа є повним циклом довжини n. Тоді ми знаємо, що кожен рядок a_i складається з послідовності символів усіх вершин, повторена тричі. Тоді для вершини v наступна вершина x_v повинна мати такий самий рядок, тільки циклічно зсунутий на один вліво. Тобто $a_{v,1} = a_{x_v,0}, a_{v,2} = a_{x_v,1}, \ldots, a_{v,0} = a_{x_v,3n-1}$.

Найти наступну вершину можна перебором.

Асимптотика рішення: $O(n^3)$.

Блоки 6-9

Візьмемо рішення для блока 4, але трохи його розвинемо. Мережа необов'язково буде деревом, але шлях кожної вершини точно закінчується в циклі (з кожної вершини є перехід і вершин скінченна кількість, тому присутність циклів гарантована). Тоді давайте спробуємо схожий підхід.

Для вершини v знайдемо вершину p, таку що $a_{v,[1,3n-1]} = a_{p,[0,3n-2]}$. У правильній мережі завжди буде існувати така вершина, але що якщо їх декілька? В такому випадку підійде будь-яка. Це все тому, що максимальна довжина циклу, в якому закінчується шлях з вершини v дорівнює n, тому якщо в рядках збігаються перші $3n-1$ символів, буде збігатися і більше.

Але що робити з невідомими рядками? Якщо в якийсь момент ми не можемо для вершини v знайти потрібну вершину, це означає, що це одна з вершин з невідомими рядками. Немає різниці, яку вершину ми виберемо, тому візьмемо будь-яку таку вершину та присвоїмо їй рядок $a_{v,[1,3n-1]}$. Але тепер маємо проблему — ми не знаємо, який має бути останній символ!

Вирішити це досить просто — нам не потрібно знати останній символ. У найгіршому випадку найкоротший рядок буде довжини $2n+1$, чого достатньо, по тій самій логіці, що і раніше — серед перших $2n$ символів будуть символи усіх вершин на шляху, включаючи повний цикл мінімум два рази. Це означає, що якщо перші $2n$ символів рядків збігаються, їх шляхи повністю однакові, тому наш алгоритм гарантовано побудує правильну мережу.

У випадку, коли усі рядки невідомі ми можемо вивести будь-яку мережу з n вершин.

Тепер, щоб більш ефективно знаходити підходящий рядок будемо рахувати хеш рядків. Тобто, для кожної вершини v додамо у список пару з номером вершини на хешу рядка $a_{v,[0,2n-1]}$. Тепер, щоб знайти x_v порахуємо хеш $a_{v,[1,2n]}$ та пройдемося по списку, шукаючи однаковий хеш.

Також існує рішення з префіксним деревом (бором) замість хешів або зі звичайним map.

Асимптотика рішення $O(n^2)$.

Задача 1C. Запити красот підмасивів

Автор задачі: **Антон Тригуб**
Задачу підготував: **Андрій Столітній**
Розбір написав: **Ігнат Жаріхін**

Введемо функцію $sign(x)$, яка повертає 1, якщо $x \geqslant 0$, та 0 інакше.

Нехай масив c розбито на k підмасивів $[c_1, \ldots, c_{p_1}], [c_{p_1}+1, \ldots, c_{p_2}], \ldots, [c_{p_{k-1}}+1, \ldots, c_{p_k}]$. Позначимо за $s_i (1 \leqslant i \leqslant k)$ суму i-го підмасиву. Тоді, краса цього розбиття — $\min(|s_1|, |s_2|, \ldots, |s_k|)$.

Якщо є такий індекс j, що $s_j = 0$, то просто об'єднаємо цей підмасив з будь-яким із сусідніх. Тоді краса розбиття зміниться з $\min(|s_i|), 1 \leqslant i \leqslant k$ на $\min(|s_i|), 1 \leqslant i \leqslant k, i \neq j$, що покращить (або, у найгіршому випадку, не змінить) відповідь.

Нехай в нас є два послідовних підмасиви, сума яких однакова за знаком. Тобто, є таке $j (1 \leqslant j \leqslant k)$, що $sign(s_j) = sign(s_{j+1})$. Об'єднаємо ці два підмасиви в один. Їх сума стане $s_j + s_{j+1}$ а краса розбиття зміниться з $\min(|s_1|, |s_2|, \ldots, |s_j|, |s_{j+1}|, \ldots, |s_k|)$ на $\min(|s_1|, |s_2|, \ldots, |s_j + s_{j+1}|, \ldots, |s_k|)$. Зрозуміло, що через те, що знаки сум s_j та s_{j+1} однакові, то $|s_j + s_{j+1}| \geqslant \min(|s_j|, |s_{j+1}|)$, а отже відповідь не погіршиться.

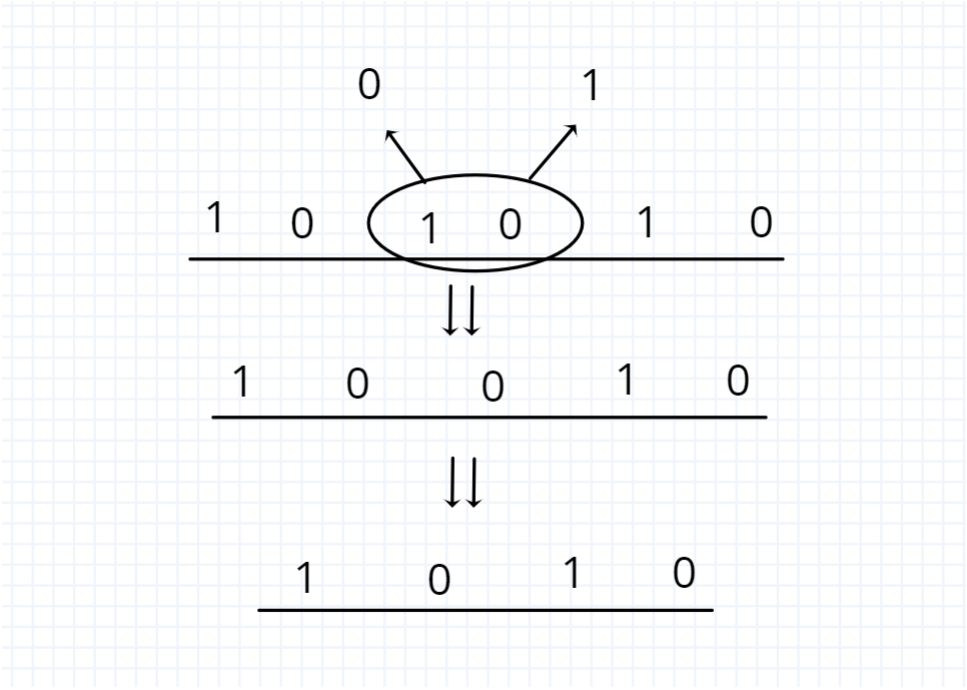
Це означає, що точно знайдеться розбиття, краса якого найбільша, а $sign(s_i) \neq sign(s_{i+1})$ для $1 \leqslant i < k$.

Блок 1

$a_i > 0$. Всі числа додатні, а отже для будь-якого розбиття масиву на підмасиви, сума елементів кожного підмасиву також буде додатньою. Тому, як ми попередньо довели, можна по черзі об'єднувати сусідні підмасиви, допоки не залишиться один, при цьому тільки покращуючи красу розбиття. Тому, відповідь для кожного запита — сума елементів від a_l до a_r. Через те, що оновлень тут немає, таку задачу можна вирішити за допомогою префіксних сум. Складність — $O(n+q)$

Блок 2

Нехай в нас є якесь розбиття на підмасиви, s_i $(1 \leqslant i \leqslant k)$ — сума і-го підмасиву, а також $sign(s_i) \neq sign(s_{i+1})$ $(1 \leqslant i < k)$. Візьмемо таке j, $1 < j < k-1$. Об'єднаємо підмасиви j та $j+1$. Тоді утвориться ситуація, як показано на малюнку 1.



Масив, розбитий на підмасиви, та один з варіантів його змінення. На малюнку 0 та 1 означають $sign()$ для відповідного підмасиву.

Помітимо, що отриманий підмасив за знаком точно дорівнює або лівому його сусіду, або правому. Об'єднаємо його з тим з сусідів, з яким знаки збігаються.

Як зміниться відповідь за всю цю операцію: спочатку в функції мінімуму було k змінних. З них, $k-3$ не змінились, 2 видалились, а 1 змінна збільшилась за модулем. Тобто, краса розбиття або стане краще, або не зміниться.

Нескладно помітити, що такі перетворення можна роботи, поки $k \geqslant 3$. Це означає, що точно знайдеться таке розбиття, краса якого — оптимальна, а кількість підмасивів в розбитті менше або дорівнює трьом.

Для кожного запиту i, розбиваємо масив на $[l_i, k_1], [k_1+1, k_2], [k_2+1, r_i]$, для цього перебираємо $l_i \leqslant k_1 \leqslant k_2 \leqslant r_i$, та рахуємо красу за допомогою префіксних сум. Складність – $O(q \cdot n^2)$

Блок 5

Нехай p — масив префіксних сум масиву a. У цьому блоці, $|p_i| \leqslant 5$. Суму будь-якого підмасиву $[l, r]$ можна представити як $p_r - p_{l-1}$. Це значить, що сума будь-якого підмасиву для цього блоку $\leqslant 10$, а отже і відповідь $\leqslant 10$.

Враховуючи те, що ми довели, що завжди оптимально розбивати на не більше ніж 3 підмасиви, то можемо для кожного запиту i перебрати s_1 та s_2, і перевірити, чи можемо ми розбити масив $[l_i, r_i]$ на три підмасиви з сумами s_1, s_2 та $(a_{l_i} + a_{l_i+1} + \cdots + a_{r_i}) - s_1 - s_2$. Для цього, треба прорахувати матрицю $d[i][s] = j, j$ — перший такий індекс, що сума елементів з i-го по j-й дорівнює s.

Також, треба не забути перевірити розбиття на 1 та 2 підмасиви аналогічним способом.

Рішення використовує $O(n \cdot C)$ пам'яті та має складність $O(n \cdot C + q \cdot C^2)$ часу, де $C = 21$.

Блок 3

Тут сказано, що існує оптимальне розбиття на не більше ніж 2 підмасиви. Випадок з розбиттям на один підмасив легко перевіряється за допомогою префіксних сум. Тому розбираємо випадок з двома масивами. Вирішимо, якщо перший підмасив має від'ємну суму, а другий додатню, а навпаки вирішується аналогічно.

Нехай m це межа наших підмасивів. Нескладно помітити, що завжди оптимально брати такий $m (l_i \leqslant m \leqslant r_i)$, що p_m — мінімальний. У такому випадку, значення другого підмасиву дорівнює $p_r - p_m$, і воно, як видно, максимізується, а отже і мінімум з модулів цих величин досягає максимуму.

Реалізувати це можна за допомогою структури даних, що вміє відповідати на запити мінімуму на відрізку, без оновлень. Будуємо структуру по масиву $p[]$. Складність – $O(q * log_2 n)$, якщо використовуючи дерево відрізків.

Блок 4, 6-9

Відповідати на запит за $O(n^2)$ ми вже вміємо — для цього треба перебирати 2 межі розбиття.

Нехай у нас є ці дві межі — k_l та k_r. Аналогічно до розбору блоку 3 можна довести, що не гірше буде взяти такі дві межі k_{lmin} та k_2, що $l_i \leqslant k_{lmin} \leqslant k_l$ та значення $p_{k_{lmin}}$ є найменшим на цьому відрізку. Аналогічно, замінюємо межу k_r на k_{rmax}, $k_r \leqslant k_{rmax} \leqslant r_i$ (Беремо максимум, адже нам треба мінімізувати третій відрізок: $p_{r_i} - p_{k_r}$).

Тобто, з початкових меж k_l та k_r ми знайшли не гірші. Через те, що на відповідь впливають лише лівий та правий відрізки, то можна ще поліпшити відповідь, якщо починати пошук не з двох меж, а з однієї.

Виходить, для якогось j : краса $= min(|minp(l_i \dots j)|, |maxp(j+1 \dots r_i) - sum(l_i \dots r_i)|)$. Перебравши j, можемо відповісти на запит за $O(n)$.

Останнє, що треба зробити, це подивитись на графіки функцій $|minp(l_i \dots j)|$ та $|maxp(j \dots r_i) - sum(l_i \dots r_i)|, l_i \leqslant j \leqslant r_i$. Обидві функції, по суті, спадають, але через наявність модулю спочатку спадають, а потім зростають. Це дозволяє використання бінарного пошуку для знаходження j.

У результаті, маємо підтримувати на вершинах дерева відрізків (побудованому по масиву префіксних сум) значення $minp$ та $maxp$. Це дає складність $O(q \cdot log_2^2 n)$. Запроси типу 2 перетворюють звичайне дерево відрізків в лініве. Аби ж вирішити задачу за $O(q \cdot log_2 n)$, треба застосувати каскадний спуск по дереву відрізків замість бінарного пошуку.

Задача 1D. AND Масив

Автор задачі: **Костянтин Савчук**
Задачу підготував: **Костянтин Савчук**
Розбір написав: **Костянтин Савчук**

Припустимо, що у вхідних даних немає нулів.

Почнемо з підзадач:

Блок 1

Потрібно реалізувати те, що написано в умові в будь-який спосіб. Наприклад переписавши псевдокод з умови на вашу мову програмування.

Блок 2

Спостереження: умова `if` буде виконуватися не більше ніж b разів, це означає, що якщо ми можемо їх швидко знаходити, то ми вирішили задачу.

Замість того щоб перебирати позицію наступного числа як в алгоритмі з умови можемо перебирати значення наступного. Серед позицій з цим значенням нас цікавить найперша яка більше вже позначених чисел. Через те, що всі числа степені 2 перебирати значення можна за $O(b)$.

Блок 3

Навчимося швидко обчислювати $f(1, p)$:

Спосіб 1:

Побудувати табличку `jump1[mask]` = найменший індекс i, що $a_i = \texttt{mask}$, або ∞, якщо не існує.

Тоді будемо перебирати значення наступного числа (повинно бути $\texttt{mask} \ \& \ x = 0$) і виберемо те, що зустрічається раніше всіх.

Можна побачити, що ми завжди беремо позиції з більшим індексом за попередній, якщо візьмемо менший, то це протиріччя, бо могли взяти на попередній ітерації.

Спосіб 2:

Побудуємо табличку `jump2[mask]` = найменше значення i, так що $(a_i \& \texttt{mask}) = a_i$ (тобто a_i є підмаскою `mask`).

Тоді наступне число має індекс $\texttt{jump2}[2^b - 1 - x]$.

Вирішуємо для всіх s. Можна перебирати значення s від n до 1 та оновлювати таблиці, так можна обчислювати $f(s, z)$ для всіх аргументів.

Асимптотика

	оновлення	пошук
спосіб 1	$O(1)$	$O(2^b)$
спосіб 2	$O(2^b)$	$O(1)$

Блок 4

Нехай $nxt = \texttt{jump2}[2^b - 1 - msk]$.

Можемо побачити що значення $f(pos, msk) = pos + f(nxt, msk | a_{nxt})$. Будемо так рахувати й зберігати значення f.

Блок 5

Можна побачити, що час роботи цих алгоритмів значно відрізняється. Виявляється, їх можна поєднати і отримати щось посередині — $2^{b/2}$ на обидві операції.

Більш загально нам потрібна структура, яка вміє робити таке: Зменшити `last(Y)` для певного значення Y до i.

Нехай $last(Y)$ — це найменший індекс такий, що $a_{last(Y)} = Y$ або 10^{100}, якщо такого немає. Обчислити $f(X) = \min_{Y \subset X} \texttt{last(Y)}$.

Розділимо бітмаску X на 2 маски: $X_{\text{low}} = X \& (2^{b/2} - 1)$ $X_{\text{high}} = X \& \sim (2^{b/2} - 1)$

Тобто X_{low} — нижні біти, а X_{high} — верхні.

Тоді можна розписати визначення f таким чином:

$$f(X) = \min_{Y_{\text{low}} \subset X_{\text{low}}} \min_{Y_{\text{high}} \subset X_{\text{high}}} \texttt{last}(Y_{\text{low}} \cup Y_{\text{high}})$$

Визначимо функцію

$$g_1(X) = \min_{Y_{\text{high}} \subset X_{\text{high}}} \texttt{last}(X_{\text{low}} \cup Y_{\text{high}})$$

Тепер маємо

$$f(X) = \min_{Y_{\text{low}} \subset X_{\text{low}}} g_1(Y_{\text{low}} \cup X_{\text{high}})$$

Якщо g перераховані, то f можна порахувати за $O(2^{b/2})$, використовуючи формулу з g. Але при оновленні `last` g також перераховується за $O(2^{b/2})$, бо одне значення y впливає лише на $g(X)$ для $O(2^{b/2})$ значень X.

Блок 6

Аналогічно

$$g_2(X) = \min_{Y_{\text{low}} \subset X_{\text{low}}} \texttt{last}(Y_{\text{low}} \cup X_{\text{high}})$$

$$f(X) = \min_{Y_{\text{high}} \subset X_{\text{high}}} g_2(X_{\text{low}} \cup Y_{\text{high}})$$

Якщо вибирати з двох визначень (через g_1 або g_2), то можна обчислювати за $O(2^{\text{popcount}(X)/2})$. Це покращить асимптотику через те, що більшість аргументів мають небагато бітів.

Задача 2A. Кольорова таблиця

Автор задачі: **Антон Тригуб**
Задачу підготував: **Валера Гриненко**
Розбір написав: **Денисов Костянтин**

Символи таблиці будемо асоціювати з кольорами, R – червоний, G – зелений, B – блакитний.

Якщо доступних кольорів лише 2, то існує лише два валідних розфарбування таблиці, які є шаховими розфарбуваннями таблиці (червоно-зелена дошка в даному випадку). Нехай c_1 та c_2 – це кількість змін клітинок таблиці, що треба зробити, щоб таблиця мала перше та друге шахове розфарбування відповідно. Оскільки колір кожної клітинки треба змінити тільки в одному з цих розфарбувань, то маємо, що $c_1 + c_2 = n \cdot m$. Тоді нескладно збагнути, що $\min(c_1, c_2) \leqslant \lfloor \frac{n \cdot m}{2} \rfloor$ і маємо, що можемо перефарбувати не більше половини клітинок, щоб не існувало пари сусідніх по стороні клітинок однакового кольору, чого і треба було досягти.

Якщо доступно три кольори, то одне з шахових розфарбувань вже не завжди підходить, тому спробуємо трохи модифікувати цю конструкцію. Розглянемо шахове розфарбування дошки у чорно-білий кольори. Спробуємо перефарбувати нашу дошку таким чином: змінюємо колір клітинок, що знаходяться на позиціях білих клітинок у чорно-білий шаховій таблиці, на червоний колір, а інші клітинки (у чорно-білий шаховій таблиці вони є чорними) перефарбовуємо у зелений або блакитний (неважливо який саме з цих) колір, якщо вони червоні. Подібних розфарбувань існує 6:

1. клітинки, на позиціях білих клітинок, фарбуємо у червоний, клітинки на позиціях чорних клітинок фарбуємо у якийсь з інших кольорів, якщо клітинка є червоною (цей варіант було описано вище);
2. клітинки, на позиціях білих клітинок, фарбуємо у зелений, клітинки на позиціях чорних клітинок фарбуємо у якийсь з інших кольорів, якщо клітинка є зеленою;
3. клітинки, на позиціях білих клітинок, фарбуємо у синій, клітинки на позиціях чорних клітинок фарбуємо у якийсь з інших кольорів, якщо клітинка є синьою.
4. клітинки, на позиціях чорних клітинок, фарбуємо у червоний, клітинки на позиціях білих клітинок фарбуємо у якийсь з інших кольорів, якщо клітинка є червоною;
5. клітинки, на позиціях чорних клітинок, фарбуємо у зелений, клітинки на позиціях білих клітинок фарбуємо у якийсь з інших кольорів, якщо клітинка є зеленою;
6. клітинки, на позиціях чорних клітинок, фарбуємо у синій, клітинки на позиціях білих клітинок фарбуємо у якийсь з інших кольорів, якщо клітинка є синьою.

Нехай R_0 – кількість червоних клітинок, що знаходяться позиціях білих клітинок у чорно-білий дошці. R_1 – кількість червоних клітинок, що знаходяться позиціях чорних клітинок у чорно-білий дошці. Аналогічно визначаємо G_0, G_1, B_0, B_1.

Тоді розпишемо скільки змін клітинок треба для кожного з 6 варіантів:

1. $G_0 + B_0 + R_1$;
2. $R_0 + B_0 + G_1$;
3. $R_0 + G_0 + B_1$;
4. $G_1 + B_1 + R_0$;
5. $R_1 + B_1 + G_0$;
6. $R_1 + G_1 + B_0$.

Покажемо, що одне з цих перефарбувань змінює не більше половини клітинок початкової таблиці. Від супротивного. Припустимо, що $G_0 + B_0 + R_1 \geqslant \lfloor \frac{n \cdot m}{2} \rfloor + 1$ і аналогічно маємо ще 5 нерівностей. Додамо всі ці 6 нерівностей. Маємо: $3 \cdot (R_0 + R_1 + G_0 + G_1 + B_0 + B_1) \geqslant 6 \cdot \lfloor \frac{n \cdot m}{2} \rfloor + 6$.

Очевидно, що $R_0 + R_1 + G_0 + G_1 + B_0 + B_1 = n \cdot m$. Тоді маємо, що $3 \cdot n \cdot m \geqslant 6 \cdot \lfloor \frac{n \cdot m}{2} \rfloor + 6 \leftrightarrow \frac{n \cdot m}{2} \geqslant \lfloor \frac{n \cdot m}{2} \rfloor + 1$. Маємо суперечність, що завершує доведення. Отже, одне з цих перефарбувань підходить. Тоді просто знайдемо, яке з них підходить і виведемо отриману таблицю.

Складність рішення: $O(n \cdot m)$.

Задача 2B. Футбол

Автор задачі: **Андрій Куц**
Задачу підготував: **Андрій Куц**
Розбір написав: **Павло Цікалишин**

Блок 1

Для кожної комбінації просимулюємо гру поки знову перший гравець буде робити нульову передачу, та знайдемо мінімальну силу гравця, що зустрівся в процесі, це і буде відповідь. Оскільки в такому випадку жодного разу не зустрінеться гравець, що двічі зробить одну й ту ж передачу, то для кожної комбінації буде максимум $n \cdot q$ кроків, тому щоб знайти силу для кожної комбінації нам буде потрібно $O(n \cdot q^2)$ операції.

Блок 2

Оскільки всі значення a_i одинакові, то відповідь завжди буде рівна першому елементу.

Блок 3

Після виконання m передач м'яч перейде до гравця що знаходиться на відстані $l = \sum_{i=0}^{m-1} k_i$ від першого і знову буде нульова передача. Щоб м'яч повернувся до першого гравця на нульовій передачі, нам потрібно зробити x раз по m передач та $x \cdot l$ повинне ділитися на n, тобто ми зробимо певну кількість повних кіл. Оскільки n просте, то або l, або x повинне ділитися на n. Якщо l ділиться на n, то нам достатньо зробити лише перші m передач і знайти мінімум серед них, тому при переході від даної комбінації до наступної нам слід зберігати та оновляти l, мінімум серед m перших передач та на якому закінчилися перші m передач. В іншому випадку x повинно бути рівним n, а оскільки всі гравці, що робили нульову передачу, повинні бути різними, то всі гравці будуть робити нульову передачу і в нашій комбінації будуть всі гравці, тобто відповіддю в даному випадку буде мінімум серед всіх гравців.

Блок 4

Хоч l може бути більшим за n, проте нам важлива лише змінна позиції в колі, тому ми можемо замінити l на $l \mod n$. Оскільки $x \cdot l$ повинне ділитися на n та x повинне бути мінімально можливим, то $x = \frac{n}{\gcd(l,n)}$. Створимо масив b, в якому ми будемо зберігати перших m гравців, що робити передачі. Тоді щоб знайти мінімальну силу гравця, що робив певну передачу, нам слід знайти мінімум серед сил гравців з номерами $b_i + l \cdot y$, де y від 0 до $x-1$. Тоді відповідь для певної комбінації буде мінімум серед відповідей для всіх передач. Щоб знаходити відповідь для певної комбінації за $O(n)$, а не за $O(n^2)$ операцій порахуємо для кожного l від 0 до $n-1$ відповідь для кожної можливої стартової позиції. Помітимо, що для кожного серед номерів $l \cdot y + 1$ відповідь буде однаковою, оскільки для початкового номера $l+1$ всі номери будуть зсунуті на 1 та номер $x \cdot y - y + l + 1$ по модулю n буде рівний 1, аналогічно, якщо початковим буде інший номер. Тому, знаючи відповідь для одного з номерів, ми можемо проставити відповідь для всіх інших зв'язаних з ним. Що дає змогу порахувати табличку за $O(n^2)$ операцій, тоді й складність алгоритму буде $O(n^2)$.

Блок 5

Якщо поглянути на табличку, що ми зробили в попередньому блоці, можна помітити, що багато рядків у ній ідентичні. Оскільки для взаємно простих l і n, $x = n$ та $(l \cdot y) \mod n$ дасть всі числа від 0 до $n-1$, то для довільного l, що є взаємно простим з n, ми отримаємо одинакові $(l \cdot y) \mod n$ числа, проте в різному порядку. Якщо l і n не є взаємно простими, то нехай $l' = \frac{l}{\gcd(l,n)}$ та $n' = \frac{n}{\gcd(l,n)}$, тоді аналогічно $(l' \cdot y') \mod n'$ дасть всі числа від 0 до $n'-1$, домноживши їх на $\gcd(l,n)$ ми отримаємо всі числа, що дасть $(l \cdot y) \mod n$. Отже, для різних l, в яких однаковий найбільший спільний дільник з n, наша табличка буде збігатися, тому ми можемо порахувати табличку лише для степенів двійки та всі l замінити на найбільшу степінь двійки, що їх ділить. Тоді, щоб порахувати табличку буде потрібно лише $O(n \cdot \log(n))$ операцій. Щоб за таку ж асимптотику знаходити силу комбінацій, слід

створити додатковий масив і для кожної степені двійки зберігати відповідь в ньому, при переході до наступної комбінації на потрібно буде лише оновити відповідне значення в масиві на значення з таблички, відповіддю буде значення в масиві для l.

Блок 6

Замінивши степені двійки на дільники числа n, в попередньому блоці ми отримаємо розв'язок до цього. Проте максимальна кількість дільників в числа рівна $\log^2(n)$, тому асимптотика буде $O(n \cdot \log^2(n))$.

Блок 7

Розв'язок попереднього блоку по часу проходить цей, проте табличка, яку ми створюємо буде завелика. Оскільки всі числа в табличці розбиті на блоки довжини x, що не перетинаються, то їхня кількість буде рівна $\frac{n}{x} = l$, також можна помітити, що перші l чисел будуть в різних блоках, а далі номера блоків будуть іти в тому ж порядку, тому ми можемо зберігати лише відповіді для перших l чисел, значення певного a_i буде рівне значенню $a_i \mod n$, яке є в нашій табличці.

Задача 2C. Герої та Монстри

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

Блок 1

Через те, що кожен герой слабкіший за кожного монстра, єдиний збір щасливих героїв — пустий. Тому відповідь дорівнює 1 для $l = 0$ та 0 для $l > 0$.

Асимптотика рішення: $O(n + q)$.

Блок 2

Нам потрібно лише знайти значення ans_1. Давайте відсортуємо усіх монстрів та героїв в порядку зростання їх сили.

Тепер будемо для кожного героя окремо перевіряти, чи ми можемо так розставити бої, щоб був щасливий тільки він. Для цього, ми повинні поставити його у парі з монстром, який буде слабкішим за нього.

Останньою проблемою залишаються інші герої — усі вони мають бути засмучені. Зауважимо, що найслабший герой має боротися з найслабшим монстром. Якщо він буде боротися з більш сильним монстром, найслабший монстр буде боротися з більш сильним героєм, який може виявитися сильніше за нього, а значить він буде щасливим, що нам не підходить.

Більш формально — якщо у нас є якийсь розподіл на бої, де усі герої засмученні та найслабший герой стоїть у парі не з найслабшим монстром, також існує розподіл, де усі герої так само засмученні, але найслабший герой стоїть у парі з найслабшим монстром. Тепер найслабший герой та найслабший монстр стоять у парі, а значить другий найслабший монстр має стояти у парі з другим найслабшим монстром (по такій самій логіці) і так далі.

Тобто, якщо у нас є множина героїв і монстрів і ми хочемо перевірити, чи існує такий розподіл, де усі герої засмученні — сортуємо монстрів і героїв за їх силою і ставимо їх у відповідні пари. Якщо усі герої засмученні — таке розбиття існує, якщо ні, то ні.

Тоді з яким монстром має боротися вибраний нами щасливий герой? Очевидно, що з найслабшим, по подібній логіці — якщо існують інші підходящі розбиття, де це не так, існує підходяще розбиття де ця умова виконується.

Тому маємо рішення — проходимо по усім героям, ставимо вибраного героя у пару з найслабшим монстром та перевіряємо, чи решта героїв будуть засмучені.

Асимптотика рішення: $O(n^2)$.

Блок 3

У цьому блоці фіксовані сили усіх монстрів та героїв. Можемо зауважити, що для кожної множини героїв без найслабшого героя існує розподіл на бої, де ця множина героїв щаслива, а усі інші — засмученні.

Показати це можна конструктивно: для найслабшого героя з множини поставимо найслабшого монстра, для другого найслабшого героя — другого монстра і так далі. Тоді кожен герой з множини буде щасливим, тому що у кожного наступного героя як мінімум на один більше монстрів, які слабкіші за нього, ніж у попереднього героя. Через те, що найслабший герой в множині — другий найслабший серед усіх (перший не може бути в множині, тому що не існує монстра, слабкішого за нього), у нас завжди буде наступний монстр, який слабкіший за нового героя.

Залишилося показати, що можна зробити усіх інших героїв засмученими. Показати це досить легко — будемо ставити найслабшого незайнятого монстра з найслабшим героєм не з множини, доки у нас залишилися незайняті монстри. Легко помітити, що кожен монстер буде сильнішим за героя.

Тоді ans_k = кількість підмножин героїв $2, 3, \ldots, n$ розміру k, що дорівнює $C_k^{n-1} = \frac{n-1!}{k!(n-1-k)!}$

Маючи ці значення, легко відповідати на запити, побудувавши, наприклад, масив префіксних сум.

Асимптотика: $O(n+q)$.

Блоки 4 та 7

Читаючи рішення, можна помітити тенденцію — якщо ми хочемо перевірити, чи ми можемо зробити множину героїв S щасливими, а усіх інших — засмученими, потрібно поставити героїв з S з найслабкішими монстрами у тому самому порядку, а інших — з найсильнішими (теж у тому самому порядку). Тоді давайте зразу відсортуємо усіх монстрів та героїв у порядку зростання їх сили.

Давайте спробуємо розв'язати цю задачу динамічним програмуванням.

Нехай $dp[i][a][k]$ — кількість таких підмножин героїв $1, 2, \dots i$, що a з них щасливі, а k — обмеження на кількість щасливих героїв. Базою буде $dp[0][0][k] = 1$ для усіх $k \in [0, n]$.

Тепер маємо всього два переходи — ми або робимо героя $i+1$ щасливим або ні. У першому випадку, ми ставимо героя $i+1$ з монстром $a+1$ (через те, що це найслабший доступний монстр) та переходимо в $dp[i+1][a+1][k]$ (такий перехід тільки можливий якщо $a+1 \leqslant k$ та герой $i+1$ сильніший за героя $a+1$).

У другому випадку ми ставимо героя $i+1$ з монстром $k+i-a+1$. Чому? Всього у нас буде k щасливих героїв, тому найслабший монстр, який буде боротися з засмученими героями — монстер під номером $k+1$. Але у нас уже є $i-a$ засмучених героїв, кожен з яких боровся з такими монстрами, тому найслабший доступний монстр — $k+i-a+1$. У такому випадку ми переходимо у $dp[i+1][a][k]$. Такий перехід можливий, тільки якщо герой слабкіший за відповідного монстра та якщо такий монстр існує ($k+i-a+1 \leqslant n$).

Таким чином ми можемо порахувати усі значення $dp[i][a][k]$. Відповідь ans_k знаходиться у $dp[n][k][k]$.

Асимптотика: $O(n^3+q)$.

Блок 5

У цьому блоці нам потрібно лише знати кількість множин S. Тоді давайте спробуємо оптимізувати попереднє рішення.

Спочатку, давайте закинемо усіх героїв та монстрів у один масив та відсортуємо його (звісно зберігаючи, хто герой, а хто — монстр). Тоді нам не потрібно окремо тримати k, щоб знати, який монстр буде найслабшим серед тих, що виграють.

Нехай $dp[i][a][t]$ — кількість множин героїв S серед перших i "персонажів"(з масиву відсортованих монстрів та героїв), таких що a з них щасливі. Тепер $t=0$ означає, що усі монстри, яких ми зустрічали, ми поставили у пару з героями сильніше за них, а $t=1$ — хоча б один монстр стоїть у парі зі слабкішим героєм. Легко помітити, що якщо ми ставимо монстра зі слабкішим героєм, усі наступні (сильніші) монстри теж мають стояти у такій парі (інакше ми порушуємо наш оптимальний спосіб розподіляти монстрів та героїв на бої).

Тому ми маємо декілька випадків та декілька переходів. Перший випадок — персонаж i це герой. У такому випадку, ми можемо спробувати зробити його або щасливим, або засмученим. Ми можемо зробити його щасливим тільки, якщо $t=0$ (інакше герой мусить програти) та якщо є вільні монстри, слабкіші за нього. Перевірити це досить просто.

Спочатку нам потрібно знати кількість вже розглянутих героїв та монстрів, нехай ці значення дорівнюють $hero_i$ та $monster_i$ відповідно. Тоді серед $monster_i$ монстрів a найслабкіших уже зайняті (через те, що вони стоять у парі з сильнішим героєм), тому ми можемо зробити героя щасливим тільки, якщо $a < monster_i$. У такому випадку ми переходимо в $dp[i+1][a+1][t]$ (не забувайте, що $t=0$ у такому переході).

Якщо ми хочемо зробити героя засмученим ми просто не робимо його поки що щасливим, тому що ми ще не зустрічали монстра, сильніше за нього. Тому просто робимо перехід у $dp[i+1][a][t]$ (у цьому переході t може дорівнювати як 0, так і 1).

Тепер розглянемо випадок, коли персонаж i — монстр.

У нас знову є два випадки. Якщо ми кажемо, що монстр буде засмученим (у парі з більш сильним монстром), то t має дорівнювати 0 (інакше більш слабкий монстр уже був щасливим, а значить і усі сильніші монстри мають бути щасливі). Тоді ми просто переходимо у $dp[i+1][a][t]$.

Якщо ж ми кажемо, що монстр буде щасливим, ми маємо спочатку перевірити, чи існує вільний

слабкий герой. Усього ми отримали $hero_i$ героїв, a з яких щасливі. Тому $hero_i - a$ з них вільні, так? Не зовсім, у випадку, коли $t = 1$, деякі з них можуть бути уже зайняті іншими монстрами, які сильніше. Через те, що у цьому блоці нам неважлива кількість щасливих героїв у множинні, давайте просто надалі використовувати a для $t = 1$. Тобто для $t = 1$, a — це кількість героїв, які уже стоять у парі, і неважливо, чи щасливі вони в тій парі, чи ні.

Тоді ми можемо зробити монстра щасливим і перейти у $dp[i + 1][a + 1][1]$.

Таким чином, ми пройдемося по усіх можливих множин щасливих героїв S, і відповідь буде дорівнювати $dp[2n][n][0] + dp[2n][n][1]$ (через те, що ми маємо розглянути усіх персонажів, кожен з n героїв під кінець має мати пару та серед усіх монстрів або буде хтось щасливий або ні).

Асимптотика рішення: $O(n^2 + q)$.

Блок 6

Попереднє рішення виглядало перспективним, але на переході, де монстр стає щасливим усе ламається. У цьому блоці ми маємо порахувати лише ans_k, тому чому б не обмежити динамічне програмування?

Якщо ми дозволимо перехід, де перший монстр стає щасливим (перехід з $dp[i][a][0]$ в $dp[i + 1][a + 1][1]$) тільки коли рівно k монстрів вже засмученні, ми під кінець будемо мати кількість тільки таких множин S, де кількість щасливих героїв дорівнює k.

Відповідь буде так само дорівнювати $dp[2n][n][0] + dp[2n][n][1]$.

Асимптотика: $O(n^2)$.

Блок 8

Попереднє рішення виглядає ще більш перспективним, але чи ми дійсно маємо окремо перебирати k і рахувати відповідь?

Виявляється, що ні. Давайте будемо рахувати два динамічних програмування замість одного. У першому ми будемо йти зліва направо (від найслабкіших персонажів до найсильніших) та будемо дозволяти монстрам буде лише засмученими. У другому ми будемо йти справа наліво (від найсильніших персонажів до найслабкіших) та будемо дозволяти монстрам лише бути щасливими.

У першому динамічному програмуванні усе так само як і в рішенні 5-го блоку, лише без переходів, де монстр щасливий. У другому усе так само, тільки тепер коли ми робимо героя засмученим нам потрібно перевірити, чи існує сильніший за нього монстр. Також i має трохи інше значення — тепер це означає, що ми роздивилися усіх персонажів, окрім i перших (найслабкіших). Ще ми навпаки міняємо a — ми починаємо з $a = n$ та зменшуємо a, коли кажемо, що герой щасливий.

Іншими словами, друге динамічне програмування — те ж саме, що і перше, тільки ми йдемо з кінця до початку і робимо усіх монстрів щасливими, а не засмученими.

Тепер загадка, як знайти ans_k? Ми знаємо, що під час переходу з $t = 0$ до $t = 1$ ми знаємо точну кількість щасливих героїв, чому б нам тоді не перебрати такий момент?

Пройдемося по усім i, таким що i-й персонаж — монстр. Тепер переберемо a — кількість щасливих героїв у момент переходу. Тоді якщо ми перемножимо $dp_1[i][a]$ та $dp_2[i][a]$ ми будемо мати кількість таких множин S, де кожен герой з S щасливий, усі інші засмученні, та $|S|$ = кількість засмучених монстрів, тобто $monster_i - 1$ (бо ми зробили усіх монстрів до i засмученими).

Таким чином ми переберемо усі варіанти, де хоча б один монстр щасливий. Варіант, де усі монстри сумні порахуємо окремо в лоб. То ж під кінець ми маємо усі значення ans_k, порахувавши лише дві ДП, тому ми можемо з легкістю відповісти на усі запити.

Асимптотика: $O(n^2 + q)$.

Бонус: розв'яжіть задачу за $O(n + \sqrt{(n \log n)}/64 + q)$.

Задача 2D. Занулити підвідрізок

Автор задачі: **Костянтин Денисов**
Задачу підготував: **Костянтин Денисов**
Розбір написав: **Костянтин Денисов**

Спершу зробимо декілька загальних міркувань, що знадобляться в рішеннях.

Навчимося рахувати $f(b)$. Розглянемо деякий масив $b_1, b_2, \ldots, b_m$. Нехай y — найстарший біт, що зустрічається у числах масиву b.

Твердження 1. Необхідно буде збільшити x (змінна з умови) так, щоб він був не менший за 2^y. Це так оскільки інакше ми жодним чином не зможемо прибрати біт y у числах, в яких він зустрічається у нашому масиві.

Тоді виходить, що всі числа, що менші за 2^y можемо "занулити"(зробити рівними нулю) використовуючи рівно одну операцію на цих числах (колись x стане рівне $b_i \leqslant 2^y$ і тоді ми використаємо операцію XOR з цим елементом, щоб зробити його рівним 0). Оскільки менше операцій, щоб "занулити"ці числа, зробити не можна, тоді маємо, що так робити оптимально.

Отже, маємо, що тепер залишилося визначити, що робити з числами, які більші за 2^y.

Твердження 2. Числа масиву, що більші за 2^y можна гарантовано "занулити"за дві операції. Дійсно, якщо маємо число $b_i > 2^y$, то можемо коли $x = 2^y$ виконати операцію XOR з b_i та коли $x = b_i \oplus 2^y$.

Нехай k − кінцеве значення змінної x після виконання всіх операцій. $k \geqslant 2^y$. Тоді можемо всі числа, що менше або рівні за k, "занулити"за одну операцію, а всі інші за дві операції. Нескладно збагнути, що за такого k оптимальніше не зробити.

Нехай $g(k)$ − кількість чисел у масиві b, що менше або рівні за k. Тоді щоб "занулити"всі числа масиву з таким кінцевим значенням x, що дорівнює k, треба виконати $k + m + (m - g(k))$ операцій: k операцій збільшення змінної x, m операцій XOR на кожне з чисел масиву та ще $m - g(k)$ операцій для чисел, що "зануляємо"за дві операції.

Тоді маємо, що треба знайти мінімум функції $k+2\cdot m-g(k)$ для $k \geqslant 2^y$. Оскільки $2\cdot m$ не залежить від k, то треба знайти мінімум функції $k - g(k)$. Нескладно довести, що мінімум досягається на деякому $k < 2^y + m$.

Блок 1. Кінцеве значення x буде дорівнювати або 2^y або a_1, де y − найстарший біт числа a_1. Тоді відповідь за запит буде $\min(2^y + 2 \cdot m, a_1 + m)$.

Блоки 2, 3. Кінцеве значення x буде дорівнювати або 2^y, де y − найстарший біт чисел з підвідрізку запита. Далі залишилось визначати скільки чисел менше або рівні за 2^y на відрізку, що можна зробити перепідрахувавши префіксні суми для кожного можливого y.

Блок 5. Достатньо було знайти мінімум функції за $k - g(k)$ за $O(n)$ при цьому відсортувавши поточний масив запиту за $O(n \log(n))$.

Блок 4. Оскільки числа на підвідрізку вже відсортовані, то це дозволяє знайти мінімум функції $k - g(k)$ за $O(\log(n))$ використовуючи дерево відрізків.

Блок 6. Створимо структуру даних, що дозволить поступово додавати елементи до неї та шукати мінімум функції $k - g(k)$, що допоможе розв'язати цей блок. Наведена далі структура даних буде використовуватись у повному рішенні.

Маємо m можливих значень $2^y \leqslant k < 2^y + m$. Створимо для кожного такого з цих значень k окрему вершину. У вершині $k = 2^y$ запишемо значення $2^y - g(2^y)$. В вершині, що відповідає значенню $k = i > 2^y$ запишемо число $i - g(i) - (i - 1 - g(i - 1)) = 1 - cnt(i)$, де $cnt(i)$ − кількість елементу i у масиві b. Між вершинами, що відповідають значенням k та $k+1$ проведемо ребро. Тобто побудували такий бамбук.

Нескладно помітити, що сума значень вершин на шляху від вершини, що відповідає $k = 2^y$, до вершини з $k = t$ дорівнює $t - g(t)$. Тоді мінімізація $t - g(t)$ еквівалентно знаходженню шляху, що проходить через вершину 2^y та мінімізує суму вершин на шляху.

Побудуємо структуру даних, що знаходить мінімум $k - g(k)$ та підтримує додавання елемента у масив b. Вважатимемо, що всі числа, що додаватимуться матимуть один і той самий старший біт y, аналогічно будуватиметься структура, що працює незалежно від того який старший біт, у числах, що додаються.

Побудуємо систему неперетинних множин (СНМ) з n вершин (n − загальна кількість чисел у всьому масиві a), де вершина i відповідає значенню $k = 2^y + i$ (тут i від 0 до $m-1$ включно). Початково в масиві немає чисел, тому значення записане в кожну вершину, крім першої, $1 - cnt(2^y + i)$ дорівнює 1. Ми намагаємось знайти шлях мінімальної суми значень вершин, що проходить через першу вершину. Неформально кажучи, якщо ми вибрали початкову вершину шляху, то ми хочемо дійти до першої вершини, але на шляху до неї наразі є "штрафні" вершини, що додають 1 до значення шляху. Отже, початково всі вершини крім першої є "штрафними". Початково $ans := \min(k - g(k)) = 2^y$ − змінна, що зберігає поточний мінімум в структурі.

Навчимося додавати нове число у структуру. Нехай додається число v, що відповідає вершині i в СНМ. Якщо ця вершина є "штрафною" , тобто значення у вершині дорівнює 1, то нове значення $1 - cnt(v)$ стане рівне 0 і вершина перестане бути "штрафною". Якщо ж вершина вже не є "штрафною" , то коли ми проходитимемо через неї, то значення на шляху зменшуватиметься чого ми й прагнемо. Спробуємо за рахунок такого "бонусу" цієї вершини прибрати іншу штрафну вершину на шляху до першої вершини, а саме знайдемо найпершу "штрафну" вершину зліва від поточної й прибираємо з неї штраф. Якщо ж такої вершини немає, то значить ми можемо дійти до першої вершини без перешкод ("штрафних" вершин), тому віднімемо від поточного мінімуму 1. Найпершу "штрафну" вершину зліва можна шукати СНМом, якщо оновлювати його відповідним чином.

Формально, для додавання нового числа й оновлення поточного мінімуму ми робимо наступне:

1. Будуємо СНМ на n вершинах та ініціалізуємо $ans := 2^y$;
2. Кожна компонента у СНМ буде якимось неперервним відрізком вершин, і підтримуємо інваріант, що найлівіша вершина у компоненті буде "штрафною" (крім найлівішої компоненти, оскільки перша вершина ніколи не є "штрафною");
3. Коли додається число v, то знаходимо компоненту в СНМ, де знаходиться вершина, що відповідає цьому числу. Також знаходимо сусідню компоненту зліва у СНМ до знайденої та об'єднуємо їх у одну, бо ми прибрали штраф з найлівішої вершини поточної компоненти;
4. Якщо ж такої сусідньої компоненти зліва немає, то віднімаємо від ans одиницю і не змінюємо жодним чином структуру СНМ;
5. ans − буде поточним мінімумом шуканої функції.

Якщо використовувати СНМ зі стиском шляху та ранговою евристикою, то можемо оновлювати структуру амортизовано за $O(\lambda(n))$, де $\lambda(n)$ — обернена функція Акермана.

Блок 7. За допомогою дерева відрізків можемо підтримувати мінімум $k - g(k)$, крім того воно буде підтримувати видалення елемента, що дозволяє використовувати алгоритм Мо з видаленнями і матимемо рішення за $O(n\sqrt{n}\log(n))$.

Блок 8. Опишемо рішення задачі, коли запити подаються в офлайні (коли всі запити відомі заздалегідь). Рішення в онлайні буде розв'язуватись схожим чином. Наступну техніку можна назвати як алгоритм Мо без видалень.

Розіб'ємо наш масив на блоки довжиною $\sqrt{n}$. Тепер відсортуємо наші запити по блоку, в якому знаходиться ліва границя запиту. Розглянемо запити, що починаються у одному такому блоці і навчимося знаходити на них відповіді. Створимо порожню структуру, що описували вище (для кожного блоку вона будується наново). Відсортуємо ці запити по правій границі, будемо відповідати на них

по її збільшенню. Нехай r – права границя блоку в якому знаходяться ліві границі запитів. Почнемо додавати до нашої структури числа масиву починаючи з $r+1$. Нехай наразі відповідаємо на запит $[x, y]$, тоді додаємо до нашої структури числа з $r+1$ до y. Оскільки y (права границя запитів) збільшується, то щоб оновлювати структуру не треба її перебудовувати кожен раз, а лише додавати нові числа, що не були додані на момент пошуку відповіді на попередній запит. Отже, зараз маємо, що майже всі числа масиву з відрізка $[x, y]$ додані в поточну структуру. Не додані лише числа з відрізка $[x, \min(r, y)]$. Таких чисел не більше ніж $\sqrt{n}$, що небагато і дозволяє обробити кожен з них окремо при пошуку відповіді на запит. На жаль, ми не можемо додавати ці числа в нашу структуру, бо тоді вона зміниться, а видаляти елементи ми не навчилися. Натомість відкатувати зміни після додавання цих елементів також не спрацює достатньо швидко, оскільки оцінка додавання в структуру є амортизованою.

Тоді треба якимось спеціальним чином обробити ці наші елементи, що не додали до структури. Розглянемо їх у порядку зменшення їх значень (щоб не сортувати масив з близько $\sqrt{n}$ чисел кожен раз, то відсортуємо елементи кожного блоку окремо і будемо проходити по всім елементам блоку в порядку спадання значень елементів і перевірятимемо чи елемент є у поточному відрізку $[x, \min(r, y)]$). Якщо додавати елементи в такому порядку, то компоненти, які ми розглядатимемо в СНМ, будуть також розглядатися в спадному порядку і тоді ми можемо зберігати вказівник на останню компоненту, що ми мали б об'єднати в СНМ, коли б додавали елемент в структуру. Фактично, ми просто моделюємо поведінку структури, не додаючи до неї елементів. Щоб моделювання було простіше робимо, то робимо це у порядку спадання елементів.

Можна було б оброблювати ці елементи інакше (схожим чином будемо робити в рішенні для онлайн запитів). А саме знайти перші $\sqrt{n}$ компонент у СНМ і далі проходитися по елементах з відрізка $[x, \min(r, y)]$ та ігнорувати елементи, що знаходяться не в перший $\sqrt{n}$ компонентах, бо вони вже не повпливають на відповідь. Тобто фактично ми робимо менший СНМ на $\sqrt{n}$ вершинах, коли відповідаємо на кожен запит і його вже змінюємо як захочемо.

Таким чином ми навчилися відповідати на запити. Оцінимо складність роботи програми: не більше $\sqrt{n}$ разів ми додамо елемент в нашу велику структуру; для кожного запиту ми ще додатково розглядаємо $O(\sqrt{n})$ елементів, тому маємо, що рішення має складність $O((n+q) \cdot \sqrt{n} \cdot \lambda(n))$.

Блоки 9, 10.

Щоб відповідати на запити в онлайні діятимемо схожим чином як офлайні (Блок 8). Оскільки ми не знаємо запити наперед, то побудуємо нашу структуру для кожного відрізку блоків (тобто беремо якийсь неперервний відрізок блоків і додаємо у структуру даних всі числа, що в ньому містяться). Тепер щоб відповісти на запит треба буде додати не більше $2 \cdot \sqrt{n}$ елементів зліва та справа від відрізка блоків. Тоді зберігатимемо $2 \cdot \sqrt{n}$ перших компонент структури і відповідатимемо на запити аналогічно до другого методу описаного в рішенні для офлайну. Складність рішення зберігається, але зараз використовуємо $O((n+q) \cdot \sqrt{n})$ пам'яті.