Разбор задачи «Лестница для участников олимпиады»

Автор и разработчик задачи — Никита Голиков

Подзадача 1.

Переберем нижний левый угол лестницы, пусть это клетка (i, j). Заметим, что в столбце j мы хотим максимизировать количество взятых клеток, так как это будет самый высокий столбец лестницы.

Будем идти по столбцам от j направо, поддерживая высоту последнего столбца лестницы. Теперь нам нужно найти максимальное количество клеток, которое можно набрать в текущий столбец, не превышая высоту последнего столбца. Сделаем это наивно за $O(h)$, получим решение за $O(h^2w^2)$.

Подзадача 2.

Ускорим предыдущее решение, ускорив часть с наивным проходом вверх по таблице. Для этого, предподсчитаем двумерный массив $\mathrm{up}_{i,j}$, который будет означать количество подряд идущих единиц в таблице, начиная с клетки (i, j) вверх. Это можно сделать динамическим программированием за $O(wh)$.

Теперь, когда мы перебираем столбец k в решении предыдущей подзадачи, мы можем за $O(1)$ вычислить высоту нового столбца, как минимум из $\mathrm{up}_{i,k}$ и высоты последнего столбца.

Получаем решение за $O(h^2w)$.

Подзадача 3.

Заметим, что $\min(h, w) \leqslant \sqrt{hw}$. Сделаем такое преобразование: повернем таблицу на 90 градусов по часовой стрелке и развернем массив строк (строка i поменяется со строкой $n - i$). При таком преобразовании количество строк и столбцов поменялось местами, а любая лестница изначальной таблицы перешла в лестницу новой, и наоборот.

Тогда решим так: если $h > w$, то сделаем вышеописанное преобразование. Теперь верно, что $h \leqslant w$, значит решение из второй подзадачи работает за $O(h^2w) = O(hw \min(h, w)) = O(hw\sqrt{hw})$, что укладывается в ограничения подзадачи.

Полное решение.

Для полного решения будем считать площадь лестницы с нижним левым углом в (i, j) с помощью динамического программирования, назовем его $\mathrm{dp}_{i,j}$. Найдем высоту первого столбца с помощью массива up, как было описано во второй подзадаче. Теперь лестница устроена так: первые сколько-то столбцов будут иметь высоту $\mathrm{up}_{i,j}$, после чего высота уменьшится. Заметим, что столбец, где первый раз высота уменьшится, это ближайший справа индекс k, такой что $\mathrm{up}_{i,k} < \mathrm{up}_{i,j}$.

Тогда верно, что $\mathrm{dp}_{i,j} = \mathrm{up}_{i,j}(k - j) + \mathrm{dp}_{i,k}$. Осталось эффективно найти значения k для каждой клетки таблицы, это стандартная задача нахождения ближайшего меньшего числа справа для массива высот в каждой строке, которая решается за линейное время проходом со стеком.

Получаем решение за $O(wh)$, которое проходит все подзадачи.

Разбор задачи «Пересменка в Сириусе»

Авторы задачи — Никита Голиков и Денис Мустафин, разработчик — Денис Мустафин

Подзадача 1.

Если какой-то номер не закреплен ни за каким сотрудником, то ответ точно «`NO`». В первой подзадаче это условие является и достаточным, достаточно направлять работников по убыванию l_i, а при равенстве по убыванию r_i. Тогда все номера, которые за кем-то закреплены будут отремонтированы.

Подзадача 2.

Здесь можно перебрать все возможные порядки направления сотрудников в номера и просимулировать процесс. Это можно сделать за $\mathcal{O}(k!kn)$.

Подзадача 3.

Здесь можно перебрать все подмножества номеров, которые уже отремонтированы по возрастанию битовой маски, поддерживая, можно ли добиться именно такого отремонтированного подмножества. Если для достижимой маски перебрать, какого сотрудника надо сейчас направить. Если обновлять маску с помощью битового или, то получится решение за $\mathcal{O}(2^n k)$.

Подзадачи 4 – 6.

Здесь нужны разные динамики по подотрезкам. Пусть $dp_{l,r}$ — 0 или 1, в зависимости от того, можно ли отремонтировать отрезок номеров с l-го по r-й и только его. Для подсчета $dp_{l,r}$ переберем

последнего работника, который что-то отремонтировал в этом сценарии. Пусть его номер i. Тогда должно быть $l \leqslant l_i$ и $r_i \leqslant r$ и до его ремонта были отремонтированы все номера на отрезках $[l, r']$ и $[l', r]$, где $l_i - 1 \leqslant r' < m_i$ и $m_i < l' \leqslant r_i + 1$ (если $l = l_i$ или $r = r_i$, то левого или правого отрезка соответственно может и не существовать). То есть должно быть $dp_{l,r'} = 1$ и $dp_{l',r} = 1$.

Если для каждого l и r перебрать все возможные i, l' и r', то получится решение за $\mathcal{O}(n^4k)$.

Заметим, что l' и r' можно перебирать независимо. Тогда получится решение за $\mathcal{O}(n^3k)$.

Вместо перебора l' надо проверить, существует ли l' в соответствующем полуинтервале, такое что $dp_{l',r} = 1$. Для этого можно параллельно с полсчетом динамики насчитывать префиксные суммы динамики по каждой из координат. То есть нас интересует $pl_{l,r} = dp_{l,r} + dp_{l+1,r} + \cdots + dp_{r,r}$ и $pr_{l,r} = dp_{l,r} + dp_{l,r-1} + \cdots + dp_{l,l}$. Тогда вместо перебора l' и r' можно проверить, что $pr_{l,m_i-1} - pr_{l,l_i-2} > 0$ и $pl_{m_i+1,r} - pl_{r_i+2,r} > 0$. Тогда дополнительно ничего перебирать не надо и решение работает за $\mathcal{O}(n^2k)$.

Подзадача 7.

Здесь за каждым сотрудником закреплен префикс или суффикс номеров. Если есть два сотрудника, за которыми закреплены префиксы номеров и они оба выполнили свои работы, то одного из них можно было не направлять (того, у кого меньше префикс). Аналогично с суффиксами. То есть достаточно вызвать только двух сотрудников: одного с префиксом и одного с суффиксом. Причем два сотрудника, отрезки которых покрывают все номера не смогут оба отремонтировать свой отрезок только если у них обоих m_i лежит в пересечении их отрезков. Назовим таких двух сотрудников *противоречащими*. То есть можно за $\mathcal{O}(k^2)$ перебрать все пары и проверить.

Подзадачи 8, 9.

Условие этих подзадач на самом деле означало, что если отсортировать отрезки по возрастанию l_i, то и l_i и r_i будут строго возрастать, а еще каждый отрезок не будет целиком покрыт двумя соседними. Назовем сотрудника *полезным*, если когда его направили в номер m_i, этот номер был не отремонтирован (и сотрудник отремонтировал весь свой отрезок). Тогда в этих подзадачах каждый сотрудник должен быть полезным. Для этого для каждой пары соседних сотрудников в порядке сортировки по l_i в пересечении их отрезков должно лежать не более одного m_i этих двух сотрудников (иначе они ни в каком порядке не смогут оба быть полезными). Это условие является и достаточным, если есть k сотрудников, то отрезков пересечения соседних не более $k - 1$, то есть для какого-то сотрудника за его m_i больше никто не ответственен и мы точно сможет направить его последним. Можно его убрать, сделать всех остальных полезными рекурсивно и после этого направить его.

То есть в этой подзадаче надо проверить, что отрезки сотрудников покрывают все номера и что соседние сотрудники не противоречат друг другу.

Подзадача 10.

Здесь тоже если отсортировать отрезки по неубыванию l_i, то r_i тоже будет неубывать, но уже не обязательно делать всех сотрудников полезными. Рассмотрим полезных сотрудников в сценарии где все номера отремонтированы. На самом деле тут тоже достаточно, чтобы никакие два соседних сотрудника не противоречили друг другу. Если какой-то отрезок целиком покрыт двумя соседними, то нетрудно видеть, что эти два соседних тоже не противоречат друг другу. То есть из того, что никакие два соседних не противоречат друг другу следует, что можно выбрать их подпоследовательность, в которой тоже соседние не противоречат и при этом выполняется условие предыдущей подзадачи.

Отсюда получается динамическое программирование: dp_i — можно ли выбрать подпоследовательность сотрудников в порядке возрастания l_j, заканчивающуюся i-м, покрывающую префикс номеров такую, чтобы никакие два соседних сотрудника не противоречили друг другу. Пересчеты можно сделать из всех i во все j с большей левой границей. Надо только проверить, что между соответствующими отрезками нет дырки и что они не противоречат друг другу. Получается решение за $\mathcal{O}(k^2)$.

Подзадачи 11 – 13.

Заметим, что если среди полезных сотрудников есть вложенные отрезки, то внутреннего из них можно было не вызывать. То есть чтобы получить общее решение достаточно в динамику из предыдущей подзадачи добавить условие, что нельзя пересчитываться во вложенный отрезок. То есть мы сортируем отрезки по l_i и пересчитываемся из i-го отрезка в j-й если верно следующее:

- $r_j > r_i \geqslant l_j - 1$
- либо $m_i < l_j$, либо $r_i < m_j$

Получается динамика за $\mathcal{O}(k^2)$. Также были менее эффективные решения с той же идеей, например $\mathcal{O}(nk)$.

Полное решение.

Теперь надо соптимизировать нашу динамику. Для пересчета в j-й отрезок подойдет любой отрезок, идущий раньше него в порядке l_i, удовлетворящий условиям из предыдущей подзадачи. То есть у него либо должно быть верно либо $l_j - 1 \leqslant r_i < r_j$ и при этом $m_i < l_j$ либо просто $l_j - 1 \leqslant r_i < m_j$. Если завести дерево отрезков на минимум, в котором после обработки i-го отрезка мы ставим на позицию r_i минимум из того, что там уже стоит и m_i, то первый вариант условия проверяется как минимум на отрезке, а второй — как проверка, что на отрезке что-то есть (то есть минимум $< \infty$). Теперь подсчет dp_j выглядит как два запроса к дереву отрезков и одно обновление. Итого решение работает за $\mathcal{O}(k \log n + n)$. Также существовало решение, проверяющее условия с помощью `std::set`.

Разбор задачи «Сочи Парк»

Автор задачи — Тихон Евтеев, разработчик — Александр Понкратов

Подгруппы с $x_0 = 0$

Если из точки x_0 пойти вправо до точки $x = x_0 + kd$, то для всех целей, чья координата не превосходит x (обозначим множество их индексов за L) оптимальное количество энергии равно $\min(x_i \,\%\, d, d - (x_i \,\%\, d))^2$. Обозначим сумму таких значений за $left$. Сумма оптимальных значений для всех координат является решением 1 подгруппы.

Так же для всех целей, чья координата больше x (обозначим множество их индексов за R) посчитаем сумму квадратов координат и сумму координат. Обозначим их за $right2$ и $right$ соответственно.

Тогда для фиксированной точки $x = x_0 + kd$ ответ выглядит следующим образом:

$$f(k) = k \cdot t + \sum_{i=1}^{n}(x - x_i)^2 = k \cdot t + left + \sum_{i \in R}(x - x_i)^2 = k \cdot t + left + \sum_{i \in R}(x^2 - 2x \cdot x_i + x_i^2) =$$
$$= k \cdot t + left + x^2 \cdot |R| - 2x \cdot \sum_{i \in R} x_i + \sum_{i \in R} x_i^2 =$$
$$= k \cdot t + left + x^2 \cdot |R| - 2x \cdot right + right2$$

Если перебрать все возможные разумные расположения участника, то получится решение за $\mathcal{O}\left(mn \cdot \dfrac{maxX}{d}\right)$, проходящее подгруппу 4.

Если предварительно отсортировать координаты, то значения $left$, $right$ и $right2$ можно считать префиксными и суффиксными суммами. Тогда переборное решение работает за $\mathcal{O}\left(m \cdot \dfrac{maxX}{d}\right)$ и проходит 6 подгруппу.

Для решения подгрупп 8, 12 и 14 заметим, что функция является выпуклой, воспользуемся тернарным поиском по количеству перемещений. Чтобы определить конкретные значения $left$, $right$ и $right2$ воспользуемся бинарным поиском или `std::lower_bound`. Получим решение за $\mathcal{O}(m \log n \cdot \log maxX)$. Также для решения подгруппы 8 существует альтернативное решение с вложенными тернарными поисками за $\mathcal{O}(m \log n \cdot \log^2 maxX)$.

Подгруппы с произвольным x_0

Очевидно, что оптимальные перемещения устроенны следующим образом: посетить несколько (возможно, ноль) точек с мячами правее x_0, а затем посетить несколько (возможно, ноль) точек с мячами левее x_0, или наоборот. При этом чтобы вернуться из части в начало, нужно потратить

столько же энергии, сколько на продвижение вперед. Поэтому можно воспринимать первую часть пути с тратой $2t$ калорий.

Заменим все координаты на $x_i - x_0$, а x_0 на 0. Пусть $A = \{x_i : x_i < 0\}$, $B = \{x_i : x_i > 0\}$. Далее можно решать задачу независимо для каждой из частей аналогично решениям, описанным выше. Итоговый ответ равен $\min(\text{solve}(A, t) + \text{solve}(B, 2t), \text{solve}(A, 2t) + \text{solve}(B, t))$, где solve — решение для $x_0 = 0$.

Полные решения

Заметим, что существует не более чем $\mathcal{O}(n)$ позиций координат, в которых меняются значения $left$, $right$ и $right2$, поэтому можно найти с помощью тернарного поиска отрезок, на котором достигается минимум, а затем найти минимум на этом отрезке вторым тернарным поиском. Асимптотика $\mathcal{O}(m(\log n + \log maxX))$. Для любого k из фиксированного отрезка все значения не изменяются и можно вместо тернарного поиска найти вершину параболы по формуле и получить решение за $\mathcal{O}(m \log n)$, которое при аккуратной реализации набирает полный балл.

Для более оптимального решения рассмотрим производную функции f

$$f'(k) = (k{\cdot}t{+}left{+}x^2{\cdot}|R|{-}2x{\cdot}right{+}right2)' = (k{\cdot}t{+}left{+}(x_0{+}kd)^2{\cdot}|R|{-}2(x_0{+}kd){\cdot}right{+}right2)' = \\ = t + 2x_0 d \cdot |R| + 2kd^2 \cdot |R| - 2d \cdot right = t + 2kd^2 \cdot |R| - 2d \cdot right$$

Теперь можно для поиска отрезка, где достигается минимум, воспользовться бинарным поиском по проиозводной и смотреть знак производной в точке k, а минимум искать формулой. Асимптотика решения также $\mathcal{O}(m \log n)$.

Разбор задачи «Лягушки на дереве»

Автор задачи — Тихон Евтеев, разработчик — Федор Ромашев

Задача требует найти максимальное паросочетание в дерве, где вершины можно брать в пару, если они находятся на нечётном расстоянии не превышающем d.

Подзадача 1.

Можно решить задачу перебором с рекурсией. Пусть текущее множество лягушек (вершин) задано как $\{v_1, v_2, \ldots, v_k\}$. Выбираем минимальную вершину и делаем два шага: либо исключаем её из множества, либо подбираем к ней подходящую вершину, находящуюся на нечетном расстоянии не более d, и удаляем обе из множества. Для проверки расстояний можно использовать любой алгоритм поиска кратчайшего пути между всеми парами вершин. В зависимости от реализации асимптотика может быть $\mathcal{O}(n!!)$ или $\mathcal{O}(2^n \cdot n)$.

Подзадача 2.

В этой подзадаче можно составлять пару из любых двух вершин, расстояние между которыми нечетное. Заметим, что дерево является двудольным графом. Можно разделить вершины на две доли, например, по четности расстояния до корня. Тогда максимальное паросочетание будет иметь размер, равный минимуму размеров долей, а пары можно восстановить, выбирая любые вершины из разных долей. Решение работает за $\mathcal{O}(n)$.

Подзадача 3.

Эта подзадача сводится к стандартной задаче нахождения максимального паросочетания в дереве. Решение можно получить либо с помощью динамического программирования по поддеревьям, либо с использованием жадного алгоритма.

Подзадача 5.

Здесь достаточно явно построить двудольный граф, в котором две вершины соединены ребром, если они находятся на нечетном расстоянии не более d. После этого можно применить алгоритм поиска максимального паросочетания в двудольном графе, например, алгоритм Куна.

Подзадачи 4, 6–8

В этих подзадачах требуется разработать алгоритм с асимптотикой $\mathcal{O}(nd)$. Основная идея — использование жадного алгоритма. При стандартном поиске в глубину мы пытаемся объединить некоторые свободные вершины из поддерева в пару. Если вершины находятся на расстояниях x и y от текущей вершины, то их можно объединить, если $x + y \leqslant d$ и они различной четности.

Однако наивный жадный подход, который объединяет все такие пары, не работает. Чтобы улучшить алгоритм, заметим, что если $x+y < d$ и вершины различной четности, то их можно будет объединить выше, а не на данной вершине. Таким образом, в каждой вершине следует объединять только те пары, у которых сумма расстояний равна d. Для этого достаточно хранить множества вершин из поддерева на каждой глубине до d от текущей вершины (например, в двусвязном списке). На каждой вершине мы жадно объединяем пары, расстояние между которыми в сумме равно d. Однако в корневой вершине необходимо запустить отдельный алгоритм жадного поиска, который для каждой вершины подбирает пару с максимально возможным расстоянием до корня.

В зависимости от реализации, можно получить решение за $\mathcal{O}(nd^2)$ или $\mathcal{O}(nd)$.

Полное решение.

Чтобы ускорить алгоритм из предыдущей подзадачи, можно использовать структуру данных для поддержки множества расстояний до текущей вершины для каждой доли, при этом хранить только те расстояния, где еще есть свободные вершины. Также необходимо эффективно определять высоту, на которой в множестве расстояний присутствует пара с суммой d. Для каждого хранимого расстояния x можно поддерживать максимальное y такое, что $x + y \leqslant d$. При этом добавлять в множество событий высоту, на которой пара даёт сумму d. При аккуратном слиянии таких структур и обновлении событий общее решение будет работать за $\mathcal{O}(n \log n)$.

Разбор задачи «Качественный отдых»

Автор задачи — Алиса Саютина, разработчики — Виктор Романенко и Иван Сафонов

В этой задаче отдельно нужно рассмотреть первую группу, когда все дни в графике выходные. Тогда при $k = 0$ или $k = 1$ ответ равен 0, а при $k \geqslant 2$ ответ равен k.

Во всех остальных случаях заметим, что каждый добавляемый выходной день всегда можно сделать днём качественного отдыха, если он будет соседствовать с каким-то другим выходным днём, поэтому каждый отгул увеличивает количество дней качественного отдыха как минимум на 1. Но если есть два изолированных выходных днях, между которыми есть один рабочий день, то взяв отгул в этот рабочий день количество дней качественного отдыха увеличивается на 3 — два существующих выходных и один новый. Разобьём все отдельные выходные дни на пары соседних, посчитаем количество таких пар n_3. Также посчитаем отдельные выходные дни, не вошедшие в эти пары n_2, и количество выходных дней, которые уже являются днями качественного отдыха.

Ответ для каждого данного k получается жадным алгоритмом. Сначала нужно выбирать отгулы между парой изолированных выходных дней, что увеличивает количество качественных дней отдыха на 3, таких отгулов может быть не более, чем n_3. Следующие отгулы будем выбирать так, чтобы они увеличивали количество дней качественного отдыха на 2, присоединяя их к изолированным выходным дням, не вошедшим в пары. Каждый такой отгул будет увеличивать число дней качественного отдыха на 2, и таких отгулов может быть не более n_2. Каждый из оставшихся отгулов увеличивают ответ на 1.

Если заранее подсчитать значения n_3, n_2 и уже существующих дней качественного отдыха, то можно отвечать за один запрос за $O(1)$ и суммарная сложность будет $O(n + q)$.

Разбор задачи «Лягушки на болоте»

Автор задачи — Павел Шишихин, разработчик — Игорь Маркелов

В задаче просят для каждой вершины графа построенного на точках ответить на вопрос: правда ли она лежит в не двудольной компоненте связности?

Подзадача 1.

Так как компонента связности не двудольная тогда и только тогда, когда в ней есть цикл нечетной длины, в этой подзадаче можно было проверить это любым полным перебором.

Подзадача 2.

В этой подзадаче можно обойти граф и раскрасить его в 2 цвета из каждой вершины за время $O(n^2)$. Граф можно было построить в явном виде.

Подзадача 3.

Здесь подойдет построение графа за $O(n^2)$ и любой обход за $O(n^2)$. Граф можно было построить в явном виде.

Подзадача 4.

Здесь требуется с оптимизировать решение из предыдущей группы по памяти. Самый простой способ это сделать — не хранить граф в явном виде.

Подзадача 5.

В этой подзадаче все точки на одной прямой. Можно показать, что в таком случае необходимо и достаточно проверить нужно ли провести ребра в 2 ближайших точки слева и справа в порядке сортировки по этой прямой. Затем обойти полученный граф за $O(n+m)$. Так как ребер будет $O(n)$ время работы решения $O(n) + O(sort)$.

Подзадачи 6-8.

В этих подзадачах можно обойти граф за $O(n \cdot r^2)$ рассматривая только точки на расстоянии не более r от текущей. В зависимости от эффективности реализация может набирать от 5 до 15 баллов.

Подзадача 9.

То что точки находятся на достаточно большом расстоянии гарантирует, что в графе линейное количество ребер. Для его построения воспользуемся следующей техникой: разобьем плоскость на квадраты со стороной $r/2$. Распределим точки по квадратам, в которые они попадают. Для каждой точки рассмотрим все точки попадающие в квадраты находящиеся на $+-r$ по x и y от квадрата в котором она находится. Так как в каждом квадрате не более 2 точек, построение графа работает за

$O(n)$ или $O(n \cdot \log(n))$ в зависимости от выбранного способа сохранения точек в квадратах. Обходить граф буде так же, как и в 5й подзадаче.

Полное решение.

Чтобы получить полный балл за задачу нужно объединить идеи предыдущих групп. Воспользуемся построением графа из 9й подзадачи и идеей из 5й подзадачи о том что достаточно проводить ребра в небольшое число соседей. Разобьем квадраты с точками на тяжелые, такие в которых хотя бы 3 точки и лёгкие, в которых менее 3. Если точка лежит в тяжелом квадрате, ответ для нее, очевидно, 1. Если точка лежит в легком квадрате, проведем из нее рёбра аналогично технике из 9й подзадачи. Так как для каждой клетки есть не более $25 * 2$ точек из которых попробуют провести ребра в точки в ней, суммарно будет проведено $O(n)$ ребер. Для того, чтобы для решения было достаточно обойти граф аналогично предыдущим подзадачам, проведем петли для всех вершин в тяжелых клетках. Итого $O(n)$ или $O(n \cdot \log(n))$ времени на построение и $O(n)$ на обход графа.

Разбор задачи «Минимизация инверсий»

Автор задачи — Алексей Михненко, разработчики — Иван Пискарев и Алексей Михненко

Обозначим за `M[a:b][c:d]` подматрицу с левым верхним углом в (a, c) и правым нижним в (b, d).

Идея динамического программирования

Пусть `dp[i][j]` – ответ для прямоугольника с левым верхним углом в (i, j) и правым нижним в (n, m).

Если первым действием была удалена первая строка, то минимальное количество инверсий в итоговом массиве это `D[i][j] + dp[i + 1][j]`, где `D[i][j]` – количество инверсий внутри `M[i:i][j:m]` плюс количество инверсий между `M[i:i][j:m]` и `M[i+1:n][j:m]` (то есть количество инверсий внутри удалённой части плюс количество инверсий, которые удалённая часть образует с оставшейся подматрицей).

Аналогично, если первым действием был удалён первый столбец, то минимальное количество инверсий в итоговом массиве это `R[i][j] + dp[i + 1][j]`, где `R[i][j]` – количество инверсий внутри `M[i:n][j:j]` плюс количество инверсий между `M[i:n][j:j]` и `M[i:n][j+1:m]`.

При известных `D[i][j]` и `R[i][j]` динамика тривиально пересчитывается за $O(nm)$.

Решение за $O(n^2m^2(n + m))$

Значения `D[i][j], R[i][j]` могут быть вычислены напрямую из определения. Для вычисления одного значения нужно перебрать пару элементов внутри удаляемой строки/столбца (не более n^2+m^2 пар для одного значения) и пару из элемента удаляемой строки/столбца и элемента оставшейся подматрицы (не более $nm(n + m)$ пар для одного значения). Всего нужно вычислить $2nm$ значений.

Итого время работы: $O(nm \cdot (n^2 + m^2 + mn(n + m))) = O(n^2m^2(n + m))$.

Решение за $O(n^2m^2)$

Можно действовать оптимальнее:

- Для вычисления числа инверсий внутри строки/столбца вычислять только разницу соседних значений. Одна разница вычисляется за линию от размера, поэтому суммарно на эту часть будет потрачено $O(nm(n + m))$ времени. Можно оптимизировать эту часть деревом Фенвика, тогда получится $O(nm \log(nm))$ времени на всю таблицу, но в этой подгруппе это не нужно.
- Подсчёт числа инверсий между удаляемой строкой/столбцом можно сделать за $O(nm)$. Достаточно насчитать массив префиксных сумм массива подсчёта одной из частей и пройтись по другой. Каждое из действий выполняется за $O(nm)$.

Решение за $O(n^2 m \log(nm))$

Оптимизируем предыдущее решение деревом Фенвика. Будем сразу считать все значения `D[i][j]`, `R[i][j]` для строки. Для этого пройдёмся вдоль длинной стороны (чтобы пересчёт работал за длину короткой), поддерживая массив подсчёта каждой из частей в дереве Фенвика и вычисляя разницу значений за O(количество добавленных элементов).

- Для `R[i][j]` пройдёмся по строкам, поддерживая остающуюся часть в дереве Фенвика, и вычисляя число инверсий удаляемого столбца проходом по нему. Суммарно будет произведено $O(n^2 m)$ запросов прибавления и $O(n^2 m)$ запросов суммы на префиксе.
- Для `D[i][j]` пройдёмся по строкам, поддерживая обе части в дереве Фенвика. После каждого добавления аналогично вычислим число инверсий, образованных только что добавленными элементами (новым элементом удаляемой строки и новым столбцом оставшейся подматрицы)

  Здесь будет использоваться два дерева Фенвика, к одному будет произведено $O(nm)$ запросов прибавления и $O(n^2 m)$ запросов суммы на префиксе, а к другому $O(n^2 m)$ запросов прибавления и $O(nm)$ запросов суммы на префиксе.

Можно добиться существенного ускорения, оптимизировав вторую часть: использовать не дерево Фенвика, выполняющее оба типа запросов за $O(\log nm)$, а корневую, которая выполняет один тип запросов за $O(1)$, а другой за $O(\sqrt{nm})$.

Идея симметричного решения

Рассмотрим разницу `R[i][j] - R[i + 1][j]`. Это в точности количество инверсий между элементом (i, j) и `M[i:n][j:m]`, плюс количество инверсий между `M[i:i][j+1:m]` и `M[i+1:n][j:j]`. Обозначим первое за `C[i][j]`, второе за `I[i][j]`.

Заметим, что разница `D[i][j] - D[i][j + 1]` тоже вычисляется аналогичным образом через `C[i][j]` и `I[i][j]`. Только вместо `I[i][j]` нужно использовать `(n - i)(m - j) - I[i][j]`, то есть количество пар, которые не образуют инверсию для `I[i][j]` (мы вычли количество пар элементов, образующих инверсию, из количества всех пар элементов).

Обратите внимание, что значений `C[i][j]`, `I[i][j]` достаточно для вычисления `R[i][j]` и `D[i][j]` за $O(nm)$. Никаких дополнительных тяжёлых (асимптотически больших $O(nm)$) вычислений производить не нужно. Причём значения `I[i][j]` считаются один раз.

Решение за $O(nm(n + \sqrt{nm}))$

- Значения `C[i][j]` можно вычислить сканлайном по значениям с 2d Фенвиком за $O(nm \log^2(nm))$
- Значения `I[i][j]` можно вычислить аналогично проходу по строкам из решения за $O(n^2 m \log(nm))$. Только на этот раз будет $O(nm)$ запросов изменения и $O(n^2 m)$ запросов суммы на префиксе, что позволяет реализовать эту часть за $O(nm(n + \sqrt{nm}))$.

Оптимизация `C[i][j]`

Заметим, что с помощью `C[i][j]` учитываются инверсии, которые будут присутствовать в итоговом массиве вне зависимости от порядка операций. Так как если клетка a была не левее и не выше клетки b, то клетка a будет идти после клетки b в итоговом массиве.

Можно вычислить суммарно число инверсий по всем таким парам клеток за $O(nm \log(nm))$:

- Выпишем табличку по строкам, по столбцам и посчитаем суммарное число инверсий в двух полученных массивах.

- Рассмотрим пару клеток a, b. Если ни одна из них не была (не строго) левее и выше другой, то в одном массиве a будет идти перед b, а в другом наоборот. Значит от этой пары клеток мы получим ровно одну инверсию к итоговой сумме. Заметим, что таких пар в точности $C_n^2 \cdot C_m^2$. Если же одна была (не строго) левее и выше другой, то если они образовывали инверсию, то эта инверсия будет учтена два раза в итоговой сумме. Значит мы можем вычислить количество таких пар, образующих инверсию, просто вычтя из общего количества посчитанных инверсий число $C_n^2 C_m^2$ и разделив полученную разницу на два. Это в точности количество инверсий, которые гарантированно (в не зависимости от порядка удаления строк и столбцов) будут в итоговой последовательности.

Решение за $O(nm\,(n + \sqrt{m}))$

Решение за $O(nm(n+\sqrt{nm}))$ имеет существенную константу, так как в нём осуществляется $O(n^2m)$ обращений к корневой, каждое из которых работает за $O(1)$, но является обращением к случайному месту в памяти.

Будем делать сканлайн по значениям в матрице. При обработке значения в клетке (i, j) хотим учесть все инверсии, которые оно образовало в `I[i-1][j], I[i-2][j], ..., I[0][j]`. Пусть `T[i][j]` – 0, если элемент (i, j) ещё не был обработан, и 1 иначе. Пусть `P[i][j]` – префиксные суммы `T[i][j]` по строкам. Тогда вклад значения (i, j) в `I[i-1][j]` это в точности `P[i-1][j]`, аналогично для `I[i-2][j], ..., I[0][j]`.

Если хранить матрицу по столбцам (то есть так, чтобы столбцы лежали последовательно в памяти), то операцию пересчёта значений `I[i-1][j], I[i-2][j], ..., I[0][j]` это поэлементное прибавление последовательного отрезка в памяти к другому последовательному отрезку в памяти. Что имеет сильно меньшую константу, чем обращение к случайному элементу (в пересчёте на элемент).

Значения `P[i][j]` можно явно поддерживать, обновляя за $O(m)$ при обработке элемента матрицы. Эта часть не может быть одновременно с предыдущей последовательной в памяти (так как это требует хранения матрицы по строкам). Но можно хранить значения `P[i][j]` в корневой, тогда обновление будет происходить за $O(\sqrt{m})$, что сделает её асимптотически легче предыдущей части при $n \approx m$.

Так как мы сделали самую асимптотически тяжёлую часть решения оптимальнее по константе, то всё решение будет работать значительно быстрее.

Решение за $O(nm\,(k \cdot n + k\sqrt[k]{m}))$

Вместо двухуровневой корневой будем использовать k-уровневую.

При почти квадратной табличке оптимально взять $k = 2$, при существенно отличающихся измерениях стоит взять большее k, например можно просто брать $k = 4$ (или использовать одно из предыдущих решений). Конкретный выбор не сильно важен, так как случай квадратной таблички самый тяжёлый.

Разбор задачи «Жизнь программистов»

Автор и Разработчик задачи — Тимофей Ижицкий

В 1-й группе $n \leqslant 20$, поэтому в ней достаточно перебрать все разбиения и найти оптимальное для каждого k.

Во 2-й группе $k = 2$. В ней можно просто перебрать разбиение за $O(n)$ и выбрать оптимальное. Но можно заметить более важное для последующих подгрупп замечание: если первый элемент максимальный, то оптимально в первый отрезок взять все элементы без последнего, а иначе оптимально взять в первый отрезок только первый элемент.

В 3-й группе $k = 3$. Достаточно перебрать первый отрезок и оптимальным образом разбить оставшийся суффикс. Для этого достаточно использовать идею из предыдущей группы.

В 5-й подгруппе можно написать динамику за $O(n^3)$, а именно пусть $dp[i][j]$ — минимальный лексикографический массив, который можно получить, разбив префикс $[a_1, a_2, \ldots, a_i]$ на j отрезков. Переходы — это или начать новый отрезок a_{i+1} элементом, или прорелаксировать максимум последнего подотрезка этим элементом.

Перейдем к ключевой идее в задаче, а именно, как для фиксированного k быстро получать оптимальное разбиение. Добавим обозначение greater$[i]$ — первая позиция $j > i$, такая что $a[j] > a[i]$. Если такой позиции нет, то greater$[i] = n$ (всё в 0 индексации). Будем жадно набирать разбиение слева направо. Пусть в текущий момент нужно разбить суффикс i на k отрезков. Если $k = 1$, то последний отрезок уже определён. Если $k = 2$, то оптимально брать отрезок $[i, min(n-1, \text{greater}[i])-1]$. Иначе, пусть в разбиение будет взят отрезок $[i, j-1]$. Так как $k \geqslant 3$ максимум на следующем отрезке равен a_j. Максимум на первом отрезке равен a_i, поэтому на j накладываются следующие ограничения:

- $j \leqslant \text{greater}[i]$, так как иначе максимум на первом отрезке будет неправильным.
- $j \leqslant n - k + 1$, так как иначе в оставшемся суффиксе нельзя будет уместить оставшийся $k - 1$ отрезок.

Таким образом, в качестве оптимального j выгодно взять минимум на отрезке от $i + 1$ до $\min(n - k + 1, \text{greater}[i])$. Это можно реализовать за $O(n \log n)$ или за $O(n)$ для каждого k, что достаточно, чтобы сдать 6-ю подгруппу. С помощью данной идеи можно также сдать 7-ю и 8-ю группы.

Далее есть два пути. В первом из них можно просто переходить от оптимального разбиения для k отрезков к оптимальному для $(k + 1)$-го. Разберёмся как именно отличаются оптимальные разбиения для k и $k+1$. Сначала у них будут совпадать все подотрезки, а потом в какой-то момент, либо жадник для k придет в состояние, когда надо брать 1 или 2 отрезка, либо граница допустимого j из текущего i увеличится на один, из-за чего можно будет получить новый минимум. Если мы берем этот новый минимум, то это означает, что все следующие отрезки будут единичной длины. Скажем, что подотрезок (переход) длинный, если его длина больше единицы, и подотрезок (переход) короткий, если его длина один. Будем следить за всеми длинными подотрезками при изменении $k+1 \to k$, какие-то длинные подотрезки удаляются, и добавляется не более одного, то есть всего их $O(n)$ для всех k.

Если сжимать все короткие подотрезки и эффективно их находить, то можно написать решение, работающее $O((n+q) \log n)$, так как сжатых коротких отрезков будет линейно. Их можно эффективно находить с помощью дерева отрезков и сетов. Но данная реализация не является самой простой.

Второй путь следующий: мы так же для всех k отдельно построим массив за $O(n)$. Для этого мы не будем искать минимум на отрезке, а будем переходить к ближайшему справа меньшему элементу, пока он левее чем greater$[i]$ и $n - k + 1$.

Теперь будем строить ответ параллельно для всех k. Для этого напишем функцию solve(pos, lk, rk), которая предполагает, что префикс перестановки до pos разбит на подотрезки и это разбиение оптимально для всех k от lk до rk. Сначала отдельно обработаем случай разбиения суффикса начиная с pos на один или два отрезка. Теперь будем параллельно эмулировать жадник для всех оставшихся k на интересном отрезке. Для этого сначала посмотрим на все возможные разрезы, перебрав цепочку ближайших справа меньших элементов. Каждый разрез оптимальный для какого-то отрезка значений k, поэтому можно запуститься рекурсивно.

Если реализовать эту идею наивно, то параллельное построение массивов для всех k будет работать за $O(n^2)$. Но можно применить следующую оптимизацию: в рекурсии можно найти максимальный возрастающий подотрезок начинающийся в pos и добавить сразу много отрезков длины 1 в сжатом виде. Структуру, умеющую добавлять сразу много отрезков длины 1 и отвечать на k-й элемент, можно реализовать обычным стеком и для ответа на запрос использовать бинпоиск. Такая оптимизация позволяет сдать 10-ю группу.

Можно заметить, что rk всегда равно $n - pos$, поэтому чтобы сдать 9-ю группу достаточно отдельно обработать случай, когда суффикс разбивается только на единичные отрезки. Замечание про rk нужно для полного решения.

Давайте заметим, что решение бы работало быстро, если бы при каждом рекурсивном запуске отрезок $[lk, rk]$ разделялся, так как таких разделений не может больше чем $n - 1$. В случайной перестановке ожидаемо каждое разделение происходит быстро, поэтому такое решение работает быстро, если эффективно обрабатывать суффикс отрезков длины 1.

Чтобы перейти к полному решению достаточно эффективно находить следующий момент рекурсии, когда отрезок $[lk, rk]$ разделится. Так как $rk = n - pos + 1$, чтобы найти ближайшее разделение, достаточно найти минимальное $j \geqslant pos$, что для некоторых k из отрезка $[lk, rk]$ будет эффективнее взять в разбиение не отрезок $[j, j]$, а отрезок $[j, \text{less}[j] - 1]$, где $\text{less}[j]$ — ближайший меньший справа элемент. Этот факт как раз следует из того, что $rk = n - pos$. После этого надо разом добавить много отрезков длины 1, после чего произойдёт разделение отрезка, что можно обработать уже явно.

Чтобы проверить, что разделение произойдёт в позиции j, надо проверить, что $\text{less}[j] < \text{greater}[j]$ и $(j - pos + 1) + (n - \text{less}[j] + 1) \geqslant lk$. Чтобы найти такое минимальное j, достаточно написать бинарные подъёмы. Это позволяет находить такое j за $O(\log n)$, что даёт асимптотику $O((n + q) \log n)$.