Разбор задачи «Расстановка ферзей»

Автор задачи и разработчик: Захар Яковлев

Разберем группы последовательно по увеличению сложности.

Ферзи, находящиеся в одном столбце, бьют друг друга, поэтому в первом тесте расстановка корректная только если в ней один ферзь, то есть $l_i = r_i$.

Во второй группе можно для каждого запроса и каждой пары ферзей в пределах отрезка проверить, бьют ли они друг друга за $\mathcal{O}(qn^2)$.

Теперь надо понять, как быстро определять, бьет ли вновь добавляемый ферзь какого-то ранее поставленного. Для этого заметим, что достаточно проверять, есть ли уже ферзь в той же строке, том же столбце или на тех же диагоналях, что и новый. Будем идти в порядке добавления ферзей и для каждой строки, столбца и диагонали (задаваемой фиксированной суммой или разностью координат) хранить номер последнего ферзя в этой линии. В группах до пятой это можно делать явно за $\mathcal{O}(m)$ памяти. Тогда при добавлении нового ферзя будем явно находить ферзя с максимальным номером, которого он бьет, массив $last$.

В третьей группе достаточно пройти по всем ферзям на отрезке и проверить, бьет ли кто-то из них ферзя с того же отрезка ($last[j] \geqslant l_i$) за $\mathcal{O}(nq + m)$.

В пятой группе все отрезки – префиксы массива, поэтому достаточно заметить, что до определенного r_i нет пары ферзей, которые бьют друг друга, а после эта пара появляется, поэтому при маленьких r_i ответ всегда «`Yes`», а при больших – «`No`». Эту границу можно найти явно за один проход, найдя первого ферзя, который бьет ранее выставленного. Итоговая асимптотика $\mathcal{O}(n + q + m)$.

В четвертой группе надо воспользоваться тем, что запросы на непрерывных отрезках. Тогда достаточно знать не последнего ферзя, которого бьет данный ферзь, но последнюю пару ферзей, которые бьют друг друга. Более того, достаточно знать номер более раннего ферзя из этой пары. Этот номер соответствует префиксному максимуму массива $last$. Теперь достаточно проверить, правда ли, что последняя пара на момент конца отрезка r_i встречается позже, чем начало отрезка l_i. Итоговая асимптотика $\mathcal{O}(n + q + m)$.

Группы 6 и 7 отличаются от групп 5 и 4 отсутствием ограничения на m. Ограничение можно обойти, если хранить только линии, на которых располагается хотя бы один ферзь. Это можно сделать, предварительно сжав координаты или храня используемые линии в map. Итоговая асимптотика в этих группах $\mathcal{O}(n \log n + q)$.

Разбор задачи «История»

Автор задачи: Илья Бурков, разработчик: Андрей Павлов

Формально в условии требуется построить граф на n вершинах, в котором присутствуют все ребра (i, j) т.ч. $a_i + a_j = S$ или $a_i \oplus a_j = X$. Ответом на задачу было совершенное паросочетание в данном графе, причем граф необязательно двудольный.

Для решения подгруппы $n \leqslant 20$ требовалось перебрать всевозможные разбиения на пары и проверить, что они подходят под условия.

В подгруппе 3 в графе не присутствовали ребра, выполняющие условия $a_i + a_j = S$. Значит подходили только пары чисел $(a, a \oplus X)$. Тогда достаточно посчитать cnt_a — количество вхождений каждого числа, после чего достаточно проверить, что $cnt_a = cnt_{a \oplus X}$ при условии, что $X > 0$.

Если же $X = 0$, то все корректные пары, в которых выполняется XOR условия это $a_i = a_j$, поэтому для подгруппы 3 достаточно проверить что cnt_a делится на 2 для каждого a. Для подгруппы 4 у нас теперь снова есть пары вида $a_i + a_j = S$, а еще мы умеем ставить в пару два одинаковых значения a, поэтому достаточно проверить, что $cnt_a \equiv cnt_{S-a} \pmod 2$, и не забыть обработать отдельно случай $2a = S$.

Далее будем считать, что случай $X = 0$ обработан отдельно и $X > 0$.

Рассмотрим подгруппу 6, в ней все a_i различны. Но тогда попробуем понять, с кем связана вершина i: из решения двух равенств можно понять, что $a_j = S - a_i$ или $a_j = X \oplus a_i$, так как все числа различны, то такое j единственно в обоих случаях. А значит в нашем графе из вершины

i исходит не более 2 ребер, иногда может исходить меньше, т.к. подобного j может не найтись в массиве. Значит наш граф выглядит как компоненты из простых циклов и путей. Для такого графа можно конструктивно найти паросочетание, использовав любой из алгоритмов обхода графов за $O(n)$.

Теперь перейдем к полному решению: в нем у нас каждое число может встречаться сколько угодно раз, поэтому построим наш граф немного иначе: запомним тот же массив cnt и оставим в массиве только различные числа, построим тот же граф и попытаемся применить решение для подгруппы 6. Главное отличие в том, что теперь у этих путей и циклов есть еще одно число на вершине cnt. По сути у нас есть операция для ребра уменьшить число на обоих его концах, такими операциями требуется сделать на всех концах значение 0.

В случае пути достаточно понять, что для конца этого пути есть число на нем x, тогда нужно взять единственное ребро с его концом ровно x раз, тогда значение в этой вершине станет 0, а в другой $y - x$ и можно будет мысленно удалить данную вершину, после чего путь уменьшится на 1 и можно будет итеративно обработать весь путь. В случае если $y - x < 0$ или осталась одна вершина с ненулевым значением, то ответа не существует.

Теперь если мы рассматриваем цикл, то в нем можно зафиксировать любую вершину и выписать от нее в какую-либо сторону цикл. После чего можно перебрать сколько раз, первое ребро на цикле будет взято, после чего можно будет мысленно забыть про это ребро и мы вернемся к случаю с путем, но это работает долго, попробуем соптимизировать. Формально на цикле записаны по порядку числа $c_1, c_2, \ldots, c_k$, и мы перебираем такое $x \leqslant \min(c_1, c_k)$, после чего рассматриваем уже путь с записанными на нем числами $c_1 - x, c_2, \ldots, c_{k-1}, c_k - x$. Тогда давайте посмотрим какие условия должны выполняться для пути, чтобы он был хорошим. $c_1 \leqslant c_2, c_2 - c_1 \leqslant c_3, c_3 - (c_2 - c_1) \leqslant c_4, c_4 - (c_3 - (c_3 - c_1)) \leqslant c_5 \ldots$ и $c_k - (c_{k-1} - (\ldots - (c_2 - c_1)) = 0$. Тогда если мы вставим сюда параметр x, получим, что для каждого четного i будет какое-то уже константное выражение $-x$, а в случае нечетных наоборот $+x$, следовательно у нас накладываются $O(k)$ ограничений на x сверху и снизу, обозначим их за $low, high$. Тогда достаточно взять любое x из отрезка $[low, high]$, подходящее под последнее условие, которое также решается достаточно легко. Из проблем в реализации существует случай $2 \cdot a_i = S$, но в данном решении оно обрабатывается достаточно просто.

Итоговую реализацию можно написать с помощью хеш-мап за $O(l_1 + l_2 + \ldots + l_m) = O(n)$ где l_i размер i-й компоненты, что заходит на 100 баллов.

Разбор задачи «Мармеладки»

Автор задачи и разработчик: Тимофей Ижицкий

Научимся решать задачу на массиве, найдем его максимальную лексикографическую подпоследовательность b. Для этого можно действовать жадно, сначала хотим делать первый элемент как можно больше. Поэтому выгодно первым элементом b_1 выбирать максимальный элемент массива, а среди всех таких самый левый. Когда мы выбрали первый элемент b, то дальше можно решать задачу на суффиксе, независимо от этого элемента. Тогда можно делать жадный алгоритм, который каждый раз выбирает наибольший элемент на суффиксе. Для решения первой подгруппы можно было реализовать это наивно, а для второй симулировать жадника при помощи дерева отрезков.

Для решения других подгрупп нужен другой взгляд на задачу. Пусть у нас есть массив, поймем, как изменится максимальная лексикографическая подпоследовательность, если приписать справа к массиву новый элемент. Рассмотрим, как будет работать описанный жадник на этих двух массивах. Сначала жадники будут выдавать одно и тоже, поскольку будет одинаковый максимум, поэтому рассмотрим первое отличие. Это отличие могло произойти только из-за добавления нового элемента, а значит он и станет максимумом, после чего жадник на новом массиве завершится. Значит, чтобы получить новую оптимальную подпоследовательность, нужно из старой максимальной лексикографической подпоследовательности удалить часть элементов на суффиксе и добавить новый элемент в конец. Пусть добавляем элемент со значением x, чтобы новая подпоследовательность была максимальна, нужно из старой максимальной лексикографической подпоследовательности удалить

все элементы на суффиксе, которые строго меньше, чем x. Удаление таких элементов будет самым оптимальным.

Тогда на самом деле такая подпоследовательность это просто все суффиксные рекорды. Благодаря этому можно решить третью и четвертую подгруппы, где нет изменений. Решаем задачу в оффлайне, хотим зафиксировать правую границу и ответить на запросы с такой границей. Тогда можно поддержать стек рекордов на суффиксе и бинарным поиском искать ответ.

Для решения шестой подгруппы можно было использовать технику с корневыми, но на полное решение ее сдать не получится. Будем делать дерево отрезков, которое ответит на все запросы. Узел ДО будет хранить

- S — размер стека рекордов в узле;
- $\mathrm{mxs}, \mathrm{mns}$ — первый и последний элементы макс лекс подпоследовательности (на самом деле это максимум и последний элемент узла);

Напишем рекурсивную функцию $\mathrm{split}(V, x)$ — длина ответа в узле V, если можно брать только элементы $\geqslant x$. Пусть L, R — левый и правый ребенок этого узла. Рассмотрим два случая

- $R(\mathrm{mxs}) \geqslant x$, то есть максимум справа хотя бы x, тогда $\mathrm{split}(V, x) = \mathrm{split}(R, x) + S - R(S)$. Если максимум справа хотя бы x, то элементы слева, которые вошли в стек рекордов узла V, должны быть хотя бы x, а значит они подходят, причем можно посчитать их количество $S - R(S)$. Для того, чтобы посчитать справа нужно просто вызваться рекурсивно.
- $R(\mathrm{mxs}) < x$, тогда максимум справа меньше x, а значит там не подойдет ни один элемент, а значит можно спокойно спуститься влево $\mathrm{split}(V, x) = \mathrm{split}(L, x)$.

Эта функция работает за $O(\log n)$ и с ее помощью мы определим все остальные. Для начала научимся мерджить два узла ДО, а именно мы знаем все значения в поддереве V, и мы хотим посчитать значения в самой V. Понятно, что мы возьмем весь стек рекордов справа у R, а слева возьмем только те элементы, которые хотя бы $R(\mathrm{mxs})$ и в стеке рекодов слева. А такое умеет говорить функция split, то есть мердж работает за $O(\log n)$. Тогда при помощи массового ДО умеем поддерживать нужные значения в узлах при массовых изменениях. Причем на один запрос изменения мы будем работать за $O(\log n^2)$.

Решим подгруппу, где $k \in \{m, m+1\}$. По сути в ней нужно было уметь определять размер стека рекордов на отрезке, поскольку последний элемент легко определяется. Сделаем функцию $\mathrm{query}(l, r, x)$ — длина стека рекордов на отрезке запроса, если можно брать элементы $\geqslant x$ и какой максимальный элемент. Она будет работать, почти как обычная функция запроса в ДО. Если нужно разделяться на два отрезка, то сначала считаем стек рекордов справа, а слева берем только элементы, большие чем максимум справа. Если отрезок попадает полностью, то надо вызывать split. Таким образом умеем отвечать на запрос за $O(\log n^2)$.

Для полного решения достаточно написать еще спуск по этим функции, для восстановления элементов. Они будут аналогичны прошлым, $\mathrm{split} - \mathrm{k}(V, x, k)$ — какой элемент k-й в стеке рекордов внутри узла V, если можно брать элементы $\geqslant x$, и $\mathrm{query} - \mathrm{k}(l, r, x, k)$ — какой элемент k-й в стеке рекордов внутри отрезка запроса, если можно брать элементы $\geqslant x$. Удобнее всего нумеровать элементы с конца. Их пересчет аналогичен предыдущим, только еще нужно понимать, в какую сторону идти при спуске. Итоговая асимптотика $O(n \log n + q \log n^2)$.

Разбор задачи «Прямоугольная квартира»

Автор задачи и разработчик: Алексей Михненко

Сначала предположим, что квартира черепашки бесконечна и не содержит мебели. Рассмотрим, какие клетки посетит черепашка, если начнёт свой путь из клетки $(1, 1)$. Если черепашка в итоге сдвинется направо больше, чем на m, то ответ на задачу 0. Иначе, для каждого y от 1 до m нетрудно видеть, что черепашка, либо не посетит ни одной клетки в столбце y, либо посетит отрезок клеток $[l_y, r_y]$: то есть все клетки (x, y), где $l_y \leqslant x \leqslant r_y$.

Величины l_y и r_y можно посчитать заранее за $\mathcal{O}(|s|)$. С помощью этих величин уже нетрудно за $\mathcal{O}(m)$ для фиксированной клетки (i, j) проверить, что она подходит под условие задачи:

- Для каждого y от 1 до m, если отрезок $[l_y, r_y]$ не пустой, надо убедится, что $j + y - 1 \leqslant m$, а так же, что все клетки в этом столбце на отрезке $[i + l_y - 1, i + r_y - 1]$ существуют и не заняты мебелью.

Чтобы для каждого столбца осуществлять эту проверку эффективно, достаточно в каждом столбце насчитать префиксные суммы. Такой способ позволит проверять фиксированную клетку за $\mathcal{O}(m)$, что даёт решение за $\mathcal{O}(|s| + nm^2)$, если сделать проверку для каждой клетки независимо. Этого решения достаточно, чтобы сдать задачу на полный балл.

Данная задача также решается за $\mathcal{O}(|s| + nm \log(nm))$ с помощью перемножения двумерного многочленов, но это не требовалось.

Разбор задачи «Простая задача»

Автор задачи и разработчик: Алексей Васильев

Изначально посчитаем вспомогательный массив $closest_{v,b}$ — расстояние до ближайшей вершины к v, число в который содержит единичный бит b ($0 \leqslant b < k$). Решим это независимо по битам. Когда мы хотим посчитать $closest$ для какого-то конкретного бита, то запустим multisource bfs из всех вершин, которые содержат этот единичный бит. Либо можно сделать это с помощью динамики по поддеревьям. Асимптотика этой части $O(nk)$.

Пусть мы знаем ответное множество A, зафиксируем в нем вершины на максимальном расстоянии. Пусть это вершины a и b. Назовем путь между этими вершинами диаметром, а длиной этого диаметра — количество ребер в нем (обозначим длину как d).

Разберем два случая:

1. Пусть длина диаметра четная. Пусть вершина m — середина диаметра. Тогда все вершины из A находятся на расстоянии не больше $\frac{d}{2}$ от m (если это не так, то можно показать, что мы взяли неправильный диаметр). Это значит, что от того, что мы возьмем в ответное множество все вершины на расстоянии не больше $\frac{d}{2}$ от m, то нам хуже не станет (то есть множество не уменьшится и стоимость этого множества останется прежней).

   Обработаем этот случай следующим образом. Зафиксируем вершину m. Теперь мы хотим найти минимальное x, что если набрать в ответное множество все вершины на расстоянии не более x от m, то у них будет нужное нам значение ИЛИ. Утверждается, что x — это максимальное значение $closest_{m,b}$ по всем битам b. Таким образом мы находим это x и релаксируем ответ через $2x$ (потому что x — это $\frac{d}{2}$).

2. Пусть длина диаметра нечетная. Тогда у него посередине не вершина, а ребро. Пусть вершины ребра посередине — это m_1 и m_2. Тогда аналогично прошлому пункту, мы говорим, что все вершины множества A находятся на расстоянии не более $\frac{d-1}{2}$ от вершины m_1 или от вершины m_2. И от того, что мы возьмем в ответное множество все вершины на расстоянии не более $\frac{d-1}{2}$ от m_1 или m_2, нам хуже не станет.

   Поэтому аналогично прошлому пункту — перебираем ребро m_1, m_2. Дальше хотим найти минимальное x, такое что, если набрать в ответное множество все вершины на расстоянии не более x от m_1 или m_2, то у них будет нужное нам значение ИЛИ. Утверждается, что x — это максимальное значение $min(closest_{m_1,b}, closest_{m_2,b})$. Находим это x и релаксируем ответ через $2x + 1$ (потому что x — это $\frac{d-1}{2}$).

   Можно еще не думать отдельно про второй случай следующим образом: добавим посередине каждого ребра фиктивную вершину с значением 0 и будем решать на таком новом дереве. После такой модификации дерева центром ответного диаметра всегда будет вершина, поэтому можно думать только про первый случай.

Асимптотика второй части $O(nk)$, потому что мы перебрали $O(n)$ кандидатов на центр диаметра и для каждого вычислили x за $O(k)$.

Разбор задачи «Минимумы на дугах»

Автор задачи и разработчик: Александр Бабин

Подзадача 1.

В этой подзадаче перестановка восстанавливается следующим образом:

$$p = [1, f(1,2), 2, f(2,3), 3, \ldots, n-1, f(n-1,n), n]$$

Подзадача 2.

В этой подзадаче можно было заранее сделать всевозможные запросы $f(i,j)$, а затем восстановить перестановку за время $O(n^2)$. Сделать это можно следующим образом:

- Пусть $g(x)$ — это количество таких y $(1 \leqslant y \leqslant n)$, что $f(x,y) = 1$.
- Ясно, что любой элемент $x \neq 1$ находится на расстоянии $\frac{n+1}{2} - g(x)$ от единицы.

Поставим единицу на позиицию $\frac{n+1}{2}$ в перестановке и произвольным образом расставим элементы, которые должны быть на расстоянии ровно 1 от нее. Так можно сделать, ввиду того, что функция f не изменяется от применения к перестановке операции циклического сдвига и разворота.

Для каждого элемента x на расстоянии $\leqslant \frac{n-3}{2}$ от единицы можно однозначно восстановить, в какой половине перестановки он находится, для этого достаточно проверить равенство $f(p_{\frac{n-1}{2}}, x) = 1$. В случае если это равенство выполняется, элемент лежит в правой половине, а иначе — в левой.

Расположение же элементов p_1 и p_n можно перебрать и проверить наивно корректность всех запросов. В некоторых случаях однозначно восстановить их взаимное расположение невозможно.

Подзадача 3.

Эта подзадача гарантировала, что решения лучше квадратичных смогут набрать хоть какие-то баллы.

Решение на 70+ баллов. (Вероятностное, сложное, но доказуемое)

Решение можно разбить на три фазы:

Фаза 1. Разбиение элементов на две половины относительно единицы.

Будем поддерживать 5 множеств элементов L — множество элементов, которые находятся левее единицы, R — множество элементов, которые находятся правее единицы, $L' \subset L$, $R' \subset R$ и C — множество элементов на расстоянии $\frac{n-1}{2}$ от единицы. Множества L' и R' будут содержать какое-то подмножество самых близких к единице элементов из множеств L и R.

Выберем случайный элемент x, для которого мы еще не смогли определить, с какой стороны он находится:

- Если $L = R = \emptyset$, то сформируем множество $R = R' = \{y \mid f(x,y) = 1\}$. В ином случае гарантируется, что $L' \neq \emptyset$ и $R' \neq \emptyset$ (см. следующий пункт) и тогда можно сформировать пару множеств $L(x) = \{y \in L' \mid f(x,y) = 1\}$ и $R(x) = \{y \in R' \mid f(x,y) = 1\}$.
  - Если $L(x) = R(x) = \emptyset$, то элемент x находится на расстоянии $\frac{n-1}{2}$ от единицы и он является *угловым*, поэтому он добавляется в множество C и следующий пункт алгоритма не производится.
  - Если $R(x) \neq \emptyset$, то $L(x) = \emptyset$. Значит, можно сформировать $R' \leftarrow R(x)$ и отнести x в левую половину.
  - Если $L(x) \neq \emptyset$, то $R(x) = \emptyset$ и надо присвоить $L' \leftarrow L(x)$.
  - Необязательно одновременно формировать множества $L(x)$ и $R(x)$, сначала можно проверить на пустоту одно, и только при необходимости сформировать второе множество.
- Положим, что мы определили, что x лежит в левой половине (т.е. в предыдущем пункте мы изменяли R'), тогда возьмем произвольный случайный элемент $y \in R'$ после обновления. И сформируем множество $L(y) = \{z \notin L \cup R \mid f(z,y) = 1\} \cup \{x\} \cup L$, присвоив $L \leftarrow L(y)$. Если $L' = \emptyset$, то присвоим $L' \leftarrow L$.
- Аналогично поступим, если x лежит в правой половине.

Таким образом если повторять два пункта выше, то значения $2, \ldots, n$ в коцне концов разобьются на 3 группы L, R, C — левые значения, правые значения и угловые. Заметим, что в ходе выполнения алгоритма множества $L(x)$, $R(x)$, $L(y)$, $R(y)$ при условии своей непустоты совпадали с $\{z \mid f(x,z) = 1\}$ или $(\{z \mid f(y,z) = 1\}$ (доказательство в качестве упражнения). Более того, можно показать, что при каждой операции величина $n - |L(x)| - |R(x)|$ уменьшалась в среднем в два раза, а также примерно в два раза уменьшалась размер L' или R'. То есть первая фаза требует $O(n)$ запросов. Разумеется, можно произвести более аккуратный анализ и строго вычислить математическое ожидание количества запросов.

Обозначим $G(x) = \{y \mid f(x,y) = 1\}$. Так как мы уже определили для некоторых элементов $G(x)$, а также для всех элементов определили с какой стороны от единицы они располагаются, то для таких элементов мы уже однозначно умеем восстанавливать их позицию. Более того, различные значения вычисленных множеств $G(x)$ для $x \in L$ позволяют разбить все элементы из y на группы подряд идущих на цикле значений. Действительно, если $G(x_1) \subset G(x_2)$, $x_1, x_2 \in L$, то все элементы из $G(x_1)$ находятся к единице ближе, чем элементы из $G(x_2) \backslash G(x_1)$.

Фаза 2. Уточнение позиций элементов из L и R.

Разобьем все элементы из R, на блоки $L \backslash G(x_1)$, $G(x_1) \backslash G(x_2)$, $G(x_2) \backslash G(x_3)$, $\ldots$, $G(x_{k-1}) \backslash G(x_k)$, где $x_i \in L$ и $G(x_k) \subset G(x_{k-1}) \subset \ldots \subset G(x_1) \subset L$, пусть это блоки $B_1, \ldots, B_k$. Для каждого блока B_i можно определить множество позиций, которые занимают значения из B_i и между разными блоками эти множества позиций не пересекаются. Более того, ясно, что если в каком-то блоке содержится ровно одно значение $x \in B_i$, позиция которого неизвестна, то ее можно сразу восстановить.

Таким образом, чем больше значений $G(x)$ $(x \in L)$ мы знаем, тем точнее можно расставить элементы из правой половины. Значения $x \in L$ точно также можно разбить по блокам, руководствуясь вычисленными множествами $G(y)$ $(y \in R)$.

Если мы выберем случайный $x \in L$, для которого еще неизвестна его позиция, то необязательно проверять все $y \in R$, чтобы сформировать $G(x)$. Во-первых, есть значения $x' \in L$, для которых мы знаем $G(x')$ и которые гарантированно находятся или левее, или правее x (исходя из деления по блокам левой половины), то из этого мы можем заключить $G(x) \subset G(x')$ или $G(x') \subset G(x)$ в зависимости от взаимного расположения. Во вторых, зная значения $G(y)$ $(y \in R)$ также можно сузить отрезок значений, для которых неизвестно, должны ли они лежать в $G(x)$ $(x \in G(y) \Leftrightarrow y \in G(x)$ + что-то известно про взаимное расположение блоков и уже поставленных на место элементов).

После этих отсечений остается только какой-то отрезок блоков из правой половины $B_l, \ldots, B_r$, для элементов которых неизвестно лежат они в $G(x)$ или нет. Сначала будут идти блоки, которые не лежат в $G(x)$, потом, блок, который частично лежит в $G(x)$, а затем блоки, которые полностью содержатся в $G(x)$. Поэтому можно воспользоваться бинарным поиском и за $\log_2(r-l)$ определить пару блоков, которые возможно, частично лежат в $G(x)$. А затем уже за сумму размеров этих блоков определяем, какой из блоков лежит частично и делим его на пару блоков.

Таким образом мы придумали достаточно экономный по запросам способ делить блоки на еще меньшие блоки. Предлагаемое решение выбирает случайно элемент x для которого неизвестно $G(x)$ (выбираются равновероятно и элементы из L и элементы из R), а затем выполняется описанная процедура разделения блоков. В конце концов все блоки делятся на блоки размера 1 и происходит победа. Разумеется, множества $G(x)$ не поддерживаются напрямую, вычисляется только разбиение на блоки.

Эту часть можно реализовать так, чтобы она работала за $O(n \log n)$ по мат ожиданию времени работы. Для этого достаточно работать за время $O(|B_x| + |B_a|)$ – где B_x — размер блока, где находится x, а $|B_a|$ — суммарный размер блоков из другой половины, которые оказались интересными для рассмотрения.

Фаза 3.

После третьей фазы восстановились все значения $p_2, \ldots, p_{n-1}$ $(p_{\frac{n+1}{2}} = 1)$, остается только расположить элементы из C (угловые элементы) на позициях p_1 и p_n. Это не всегда можно сделать однозначно, но утверждается, что достаточно рассмотреть значения $f(c, p_2)$ и $f(c, p_{n-1})$ $(c \in C)$ и по ним восстановить любое подходящее расположение угловых элементов.

Не легко, но плюс-минус интуитивно, можно показать, что математическое ожидание количества запросов у такого решения не превышает $O(n \log n)$.

Решение на 85+ баллов. (Теория информации)

Пусть мы находимся на второй фазе алгоритма. По сути, уже определены C — угловые элементы (на расстоянии $\frac{n-1}{2}$ от единицы), определены позиции некоторых элементов, а остальные элементы разбиты на блоки $B_1, \ldots, B_k$, где каждый блок состоит из каких-то значений, для которых известно, что они находятся на каком-то множестве подряд идущих значений, на которых еще не стоят восстановленные элементы перестановки (какой-то найденный элемент при этом может стоять между какими-то двумя возможными позициями для одного блока), при этом эти множества позиций не пересекаются.

Тогда, имея эту информацию возможно $2^J = |B_1|! \cdot |B_2|! \cdot \ldots \cdot |B_k|!$ возможных перестановок, количество информации J вычисляется, соответственно, как $J = \sum_{i=1}^{k} \log_2 |B_i|! = \sum_{i=1}^{k} \sum_{j=1}^{|B_i|} \log_2 j$. Во многих задачах применима эвристика на основе теории информации — достаточно выбирать жадную стратегию, которая на каждом шаге максимизирует отношение уменьшения количества информации к количеству потраченных запросов, т.е. максимизируется математическое ожидание величины $\frac{-\Delta J}{\Delta q}$ по всем допустимым шагам, где ΔJ — это количество информации после очередного шага, а Δq — потраченное на этот шаг количество запросов.

В нашем случае, шаг заключается в том, что мы выбираем какой-то элемент x с неизвестной позицией, а затем находит для него $G(x)$ ($G(x) = \{y \mid f(x, y) = 1\}$), после чего узнаем позицию элемента x, а также разрезаем какой-то блок на две части.

Рассчитаем количество запросов и уменьшение количества информации, которое произойдет, если мы определим позицию элемента с номером i:

- $-\Delta J = \log_2 |B(i)| - \log_2(X!) - \log_2(Y!) + \log_2((X+Y)!)$, где $|B(i)|$ — размер блока, в котором содержится элемент i, а X и Y — это размеры блоков, на которые разрезается блок, после нахождения блока i.
- $\Delta q \approx X + Y + \log_2 M(i)$, где X и Y — те же величины, что и в прошлом пункте, а $M(i)$ — количество блоков, которые может разрезать элемент x', находящийся в том же блоке, что и x.

Соответственно, математическое ожидание отношения полученной информации к затраченным действиям $E[-\frac{\Delta J}{\Delta q}]$ для конкретного блока B_j равно средне арифметическому $\frac{-\Delta J}{\Delta q}$ для всех возможных индексов $i \in B_j$, где могут лежать элементы из B_j. И жадная стратегия будет заключаться в том, чтобы взять случайное значение из того блока, в котором это матожидание как можно больше.

Остается только научиться пересчитывать все описанные значения. Удивительно, но это все можно делать наивно, суммарное количество изменений ΔJ и Δq по всем индексам оказывается не очень большим.

Решение на 100+ баллов. (Дерево минимумов)

Предположим, что мы разделили все элементы на 3 группы L, R, C (читать первую фазу алгоритма на 70 баллов). Тогда можно потратить $n \log_2 n$ запросов на построение дерево миниумов по элементам L и по элементам R. Деревом минимумов называется дерево, которое строится для массива $[a_1, \ldots, a_n]$ следующим образом:

- Выбирается минимальный по значению элемент a_i, он становится корнем дерева, а левым и правым сыновьями этой вершины становятся деревья минимумов, построенные для подотрезков $a[1 \ldots, i-1]$ и $a[i+1, \ldots, n]$.

Легко заметить, что если мы выбираем два значения $x, y \in L$, то $f(x, y)$ будет равняться LCA вершин x и y в дереве миниумов для подотрезка перестановки, где располагаются элементы из L.

Для построения дерева минимумов для множества L можно постепенно добавлять в него элементы в порядке возрастания. Пусть мы добавляем элемент x, тогда можно выделить путь из корня в уже построенном дереве, который всегда идет в большее поддерево. Если этот путь заканчивается в листе y, то после запроса $f(x, y)$ мы понимаем, LCA вершин x и y, то есть тот момент, в который надо сойти с этого пути. Сошли с пути — зашли в поддерево в два раза меньше, можем повторить запрос, за $O(\log n)$ запросов смогли понять, куда подвесится вершина x. Единственным нюансом

является тот факт, что в построенном дереве непонятно, какие сыновья левые, а какие правые, но даже без этой информации построенное дерево помогает ускорить вторую фазу алгоритма.

Заметим, что когда разрежется какой-то блок или из блока мы узнаем значение какого-то элемента и его понадобится вырезать из дерева минимумов, то мы все спокойно сможем пересчитать и не будет никак проблем. Достаточно разрезать исходное дерево минимумов на минимальное количество деревьев, а потом корни полученных деревьев соединить в одно дерево тем же алгоритмом.

Тепрь рассмотрим то, почему дерево минимумов в принципе может помогать. Пусть у нас есть элемент x и набор блоков с построенными деревьями $B_1, \ldots, B_n$, а в данный момент мы разрезаем элементом x какой-то блок B. Обозначим $a(v) = 1$ ($v \in B$), если $f(x, v) = 1$ и $a(v) = 0$ в ином случае. Есть следующие отсечения:

- Если v — корень, а l и r — какие-то вершины из его разных поддеревьев, то если $a(v) \neq a(l)$, то $a(v) = a(r)$.
- Если v — корень, а в одном из его поддеревьев есть y ($a(y) \neq a(v)$), то можно отделить от дерева v сына, в поддереве которого лежит вершина y, и однозначно восстановить какое дерево левее, а какое правее (то есть дополнительно разделить все на блоки).
- Вычислять значения $a(v) = f(x, v)$ выгодно в порядке убывания размеров поддеревьев v.

Эта пара отсечений работает хорошо, так как разрезает блок, не на один блок, а по крайней мере на $\Omega(\log(n))$ по матожиданию, если разрезать блоки в случайных местах. Более того, такой способ дает некоторые гарантии (сейчас будет рукомахание, но все же):

- Мы режем дерево в случайном месте, поэтому чаще всего $f(v, x) = 1$ и $f(v, x) \neq 1$ будет встречаться примерно с соизмеримой вероятностью. Поэтому будет очень часто работать отсечение по первому пункту и мы сделаем мало запросов на сбалансированном дереве.
- Если же мы режем дерево не в равной пропорции, то мы сделаем много запросов, но и разрежется много ребер, потому что мы достаточно глубоко спустимся.

Разбор задачи «Титаник»

Автор задачи и разработчик: Александр Заварин

Для начала поймем, что, так как крюк может выстреливать только перпендикулярно движению лодки, то существует всего 1 точка на прямой, проведенной через точки A и B. Это значит, что мы можем вычислить отрезок на прямой, на котором крюк будет заблокирован, если мы зацепим шлюпку в начальной точке. Для того чтобы вычислить точку, в которой нужно зацепить шлюпку, нужно провести перпендикуляр к прямой AB, основание перпендикуляра и будет нужной точкой. Чтобы вычислить точку, где крюк освободится, необходимо из начальной точки провести вектор в направлении движения спасательной лодки с длиной, равной длине перпендикуляра. Будем называть для шлюпки начальную точку этого отрезка — *точкой захвата*, а конечную точку — *точкой спасения*.

Рассмотрим 1 подгруппу, в ней есть ограничение на координату y у точек A и B, она у них равна 0. Б. о. о. пусть спасательная лодка плывет слева направо (иначе можно просто отзеркалить координаты относительно прямой $x = 0$). Тогда, если i-я шлюпка находится в точке (x_i, y_i), то точка захвата у нее имеет координаты $(x_i, 0)$, а точка спасения — $(x_i + y_i, 0)$, т.к. расстояние между точкой захвата и шлюпкой было ровно y_i. Мы можем отбросить все шлюпки, у которых x-координата точки захвата меньше a_x или больше b_x, потому что одни мы не сможем захватить крюком, а другие не успеем притянуть к себе — игра закончится. Теперь отсортируем все шлюпки по x-координате точки захвата, чтобы воспользоваться динамическим программированием. Посчитаем для каждой шлюпки i величину dp_i — максимальное число шлюпок среди тех, у которых номер в отсортированном порядке меньше i, и при этом спасти i-ю шлюпку. Теперь просто одним циклом будем перебирать

номер лодки в отсортированном порядке, пусть сейчас это i, а теперь переберем все $j < i$, для j-й лодки проверим, что ее x-координата точки спасения не больше x-координаты точки захвата нашей i-й лодки, иначе мы не сможем спасти ее. Вычислим $dp_i = \max_{j<i}(dp_j) + 1$ среди подходящих j. Это работает, потому что для dp_i среди спасенных лодок с номерами $j < i$ есть лодка с максимальным номером j_{max}, и для нее, если убрать спасенную i-ю лодку, $dp_{j_{max}} = dp_i - 1$ по определению dp_i. Тогда, так как ответ тоже содержит в себе некоторую лодку i_{max} с наибольшим номером, то ответ — это $dp_{i_{max}}$, т.е. его можно вычислить как $\max_i(dp_i)$.

Перейдем ко 2 подгруппе, теперь отсортировать лодки так удобно не выйдет, однако это можно сделать, используя теперь в качестве параметра сортировки расстояние между точкой A и точкой захвата у шлюпки. Это можно вычислить, используя формулы построения перпендикуляра к прямой и вычисления расстояний между точками, однако авторское решение использует векторные и скалярные произведения для подсчета этого расстояния. Пусть шлюпка находится в точке P, тогда расстояние от A до точки захвата можно вычислить как $|AP| \cdot \cos\alpha$, где α — это угол между AP и AB. Это можно выразить через скалярное произведение векторов как $|AP| \cdot \cos\alpha = \frac{AB \cdot AP}{|AB|}$. Длина перпендикуляра рассчитывается подобным образом, но через векторное произведение: $|AP| \cdot \sin\alpha = \frac{|AB \times AP|}{|AB|}$, важно не забыть взять модуль от векторного произведения: в первом случае модуль не использовался, чтобы шлюпки вне доступности спасательной лодки имели отрицательное расстояние от точки A для удобства. Теперь, используя эти две величины, мы можем посчитать расстояние от точки A до точки спасения шлюпки как их сумму. Заметим, что у расстояний общий знаменатель — длина отрезка AB. Давайте тогда просто домножим все величины на это число, от этого порядок лодок и их сортировка не изменятся, но зато мы сможем перейти к вычислениям в целых числах. Теперь, имея две величины для каждой шлюпки — расстояние от точки A до точки захвата и точки спасения, а также используя идею с динамическим программированием из 1 подгруппы, можем сдать эту.

Перейдем к 3 подгруппе, и теперь подумаем, как улучшить подсчет величины dp_i. Для этого воспользуемся *scanline*-идеей, у нас будет 2 типа событий: “мы прибыли в точку захвата некоторой шлюпки” и “мы прибыли в точку спасения некоторой шлюпки”. Мы отсортируем все события по расстоянию от точки A до них, теперь будем идти по ним и хранить ответ — максимальное число лодок, которое мы можем спасти, если мы прошли первые i событий, обозначим его $score$. Если мы встречаем событие 1-го типа для j-й шлюпки, то мы вычисляем $dp_j = score + 1$ по определению. Если мы встречаем событие 2-го типа для j-й шлюпки, это значит, что, если бы мы эту шлюпку спасали, то крюк бы освободился, а значит, мы можем теперь использовать dp_j для обновления $score$: $score = \max(score, dp_j)$. Важно, что, если несколько событий находятся в одной точке, то сначала нужно обработать события 2-го типа, а потом уже первого типа. После обработки всех событий $score$ и будет нашим ответом.

Подгруппы 4 и 5 используют объединенные идеи подгрупп 2 и 3.

Разбор задачи «Играем в го»

Автор задачи: Алексей Михненко, разработчик: Игорь Маркелов

Для начала посмотрим на оптимальный ответ. Очевидно, что площадь ответа будет не меньше, чем площадь выпуклой оболочки исходного множества точек. В случае, если площадь больше, утверждается, что белая точка которую мы подвинули сдвинута относительно нормали к одному из $n \cdot (n-1)/2$ отрезков соединяющих исходные точки.

Это наблюдение дает нам возможность написать решение за $O(n^4 \cdot \log(n))$ — переберем направление и точку, сдвинем, пересчитаем оптимальный ответ через площадь выпуклой оболочки полученного множества.

Заметим (разумеется, доказательство этого потрясающего наблюдения остается в качестве интереснейшего упражнения для любознательного читателя), что для каждого ориентированного направления достаточно рассмотреть не более 5 точек (наиболее удаленных по этому направлению). Таким образом предыдущее решение превращается в $O(n^3 \cdot \log(n))$, если для каждого направления в начале выбирать не более 5 кандидатов, а затем для каждого из них строить выпуклую оболочку.

Научимся для зафискированного множества из k точек отвечать на запрос «площадь выпуклой оболочки множества, если добавить в него еще одну точку» за $O(\log(k))$ с предподсчетом за $O(k \cdot \log(k))$. Построим выпуклую оболочку на k точках, посчитаем префиксные суммы площадей выпуклой оболочки от первой вершины до i-й для всех вершин. Для ответа на запрос, в начале стандартным алгоритмом за $O(\log(k))$ проверим, лежит ли точка внутри выпуклой оболочки. В случае, если лежит — ответ равен площади исходной оболочки. В случае, если не лежит — найдем касательные из точки к выпуклой оболочке, площадь ответа будет равна площади треугольника из точек касание и точки для которой мы отвечаем на запрос к которой нужно прибавить площадь исходной выпуклой оболочки без части многоугольника отрезаемой диагональю между точками касания. Поиск касательных можно реализовать за $O(\log(k))$, а необходимые площади вычислить за $O(1)$ воспользовавшись предподсчитанными префиксными суммами.

Применим этот примитив к предыдущему решению, предподсчитаем выпуклые оболочки исходного множества без каждой из белых точек, тогда решение будет работать за $O(n^3)$, так как самая тяжелая часть это рассмотрение $O(n)$ точек-кандидатов для $O(n^2)$ направлений.

Внимательный читатель условия, дойдя до этого места в своих рассуждениях, возможно оценил щедрый жест со стороны жюри — убрать из рассмотрения тесты с тремя и более точками на одной прямой.

Оптимизируем эту часть с помощью стандартной техники вращающегося сканлайна по направлениям. Для $O(n^2)$ направлений требуется поддерживать отсортированный вдоль направления список из $O(n)$ точек. Суммарное количество событий будет $O(n^2)$, соответственно теперь решение будет работать за $O(n^2 \cdot \log(n))$.

Разбор задачи «Переезд»

Автор задачи: Александр Бабин, разработчик: Иван Пискарев

$O(S! \cdot S)$

Переберём все возможные способы переезда и выберем лучший.

$O(S^3 \cdot \log m)$

Будем искать ответ бинарным поиском. Хотим проверить, что можно организовать переезд так, чтобы максимальное расстояние было не больше k. Построим двудольный граф, где вершины в левой доли соответствуют местам до переезда, а в правой местам после переезда. Ребро между двумя вершинами присутствует, если расстояние между их домами не превосходит k. Несложно видеть, что переезд с расстоянием не больше k осуществить возможно тогда и только тогда, когда в графе существует полное паросочетание.

Найти максимальное паросочетание в двудольном графе можно алгоритмом Куна за $O(nm)$, где n – число вершин, m – число рёбер.

$O(2^A \cdot AB)$

Вместо явного поиска максимального паросочетания достаточно лишь проверить условие леммы Холла. А именно, проверить, что с каждым подмножеством X вершин левой доли напрямую связано хотя бы $|X|$ вершин правой. Так как такое множество вершин правой доли зависит только от множества клеток, в которых находятся вершины множества X, то достаточно рассмотреть всего 2^A вариантов такого множества.

$O(m)$ **при** $n = 1$

Упорядочим места до переезда и места после переезда по возрастанию j-координаты. Заметим, что способ переезда, где жилец из k-го места до переезда переезжает в k-ое место после переезда,

оптимален. Найти максимальное расстояние при таком способе переезда можно проходом с двумя указателями.

$O(m \cdot \log m)$ при $n \leqslant 2$

Сделаем бинпоиск по ответу. Хотим проверить, можно ли организовать переезд так, чтобы максимальное расстояние было не больше k. Для этого проверим условие леммы Холла.

Пусть X – подмножество домов.

- Обозначим за $W_1(X)$ и $W_2(X)$ – суммарное количество жителей в X до и после переезда соответственно.
- Обозначим за $N(X)$ – множество домов на расстоянии не более k от X.

Тогда условие леммы Холла равносильно выполнению неравенства $W_2(N(X)) - W_1(X) \geqslant 0$ для всех X. Чтобы это проверить, найдём минимальное значение $f(X) = W_2(N(X)) - W_1(X)$ и сравним его с нулём.

Для нахождения минимального значения воспользуемся динамическим программированием: пусть `dp[i][mask]` – минимальное значение $f(X)$, если

- X состоит только из домов до i-го столбца включительно
- X содержит хотя бы один дом в i-ом столбце
- `mask` – пересечение X с i-ым столбцом

Заметим, что если X содержит хотя бы один дом в i-ом столбце, то вхождение дома строго правее i-го столбца в $N(X)$ не зависит от вхождения в X домов строго левее i-го столбца и наоборот. Значит для пересчёта достаточно перебрать ближайший справа дом, который добавится в X.

Заметим, что достаточно пересчитываться только в столбцы на 1 и $2k$ домов правее. Так как при пересчёте на не больше чем $2k - 1$ столбцов включение всех пропущенных столбцов в X не сделает хуже, а значит такой пересчёт будет не хуже нескольких пересчётов на один столбец вправо. И при пересчёте больше чем на $2k$ столбцов при значении `dp` не меньше 0 его не выгодно использовать, а при значении меньше нуля уже был найден контрпример к условию леммы Холла.

$O(m)$ при $n \leqslant 2$

От бинпоиска можно избавиться. Заметим, что от склейки первой и второй строк ответ либо не изменится, либо уменьшится на один (так как расстояние между любыми двумя местами либо не изменилось, либо уменьшилось на один). Значит достаточно решить такую же задачу для на один меньшего n и после сделать проверку для одного значения k.

Полное решение $O(mn^2 \cdot 3^{2n})$

Пусть у нас есть $\mathbb{N}$ красок различных цветов, пронумерованных числами $0, 1, 2, ...$

Введём несколько обозначений:

- Для множества домов X определим раскраску домов $C(X)$, в которой дом покрашен в цвет с номером, равным расстоянию до ближайшего дома из X.
- Для раскраски домов C определим множества домов $X(C), Y(C)$, состоящие из домов цвета 0 и домов цветов $0, 1, 2, ..., k$ соответственно.
- Определим стоимость раскраски C как $f(C) = W_2(Y(C)) - W_1(X(C))$.

- Назовём раскраску хорошей, если разница номеров цветов любых двух соседних домов не превосходит 1.

Заметим, что

- $f(X) = f(C(X))$
- $C(X)$ – хорошая раскраска
- Если C – хорошая раскраска, то $f(C) \geqslant f(X(C))$

В предыдущем решении мы сводили задачу к вычислению $\min\limits_{X} f(X)$.

Заметим, что из трёх свойств выше следует, что $\min\limits_{X} f(X) = \min\limits_{C \text{ хорошая}} f(C)$.

Значит достаточно научиться вычислять $\min\limits_{C \text{ хорошая}} f(C)$.

Для этого пересчитаем `dp[i][mask]` – минимальное значение $f(C)$ по всем хорошим раскраскам C, если

- В W_1, W_2 учитываются только жители домов до i-го столбца включительно
- `mask` кодирует номера цветов домов i-го столба

Пересчёт за $O(m^2 n \cdot 3^{2n})$

Попробуем сначала просто пересчитывать `dp[i+1]` через `dp[i]`.

Заметим, что не имеет смысла рассматривать раскраски, в которых нет цвета 0, значит максимальный цвет не превосходит $n + m = O(m)$. Заметим, что если цвет первого дома в столбце фиксирован, то оставшиеся можно раскрасить не более 3^{n-1} способами, чтобы при этом раскраска получилась хорошей.

Значит возможных значений `mask` не более $O(m \cdot 3^n)$.

Заметим, что из `dp[i][mask]` можно сделать пересчёт не более чем в 3^n состояний в `dp[i+1]`.

Один пересчёт из `dp[i][mask1]` в `dp[i+1][mask2]`, очевидно, можно сделать за $O(n)$.

Итого у нас есть $O(m^2 \cdot 3^n)$ состояний `dp`, из каждого из которых мы делаем не более 3^n пересчётов, каждый из которых работает за $O(n)$. Значит время работы такого решения это $O(m^2 \cdot n \cdot 3^{2n})$

Заметим, что достаточно рассматривать только раскраски из цветов $0, 1, ..., \mathrm{k}, \mathrm{k}{+}1$. Это позволяет решить задачу за $O(\mathrm{ans} \cdot mn \cdot 3^{2n})$.

Пересчёт за $O(mn^2 \cdot 3^{2n})$

Будем явно хранить только те состояния, где в `mask` присутствует цвет 0. Заметим, что таких значений `mask` ровно 3^{n-1} (выбираем $n-1$ разностей соседних из $\{-1, 0, +1\}$ и прибавляем константу так, чтобы минимум был равен нулю).

Заметим, что достаточно пересчитываться только на $1, 2, ..., 2n$ или $2k - 2(n-1), ..., 2k-1, 2k$ столбцов направо. Пусть мы пересчитались из столбца i_1 в столбец i_2. Если $i_2 - i_1 < 2k - 2(n-1)$, то, добавив столбцы с $i_1 + n$ по $i_2 - n$ в множество X, мы не увеличим стоимость, так как $Y(C)$ останется прежним. Если же $i_2 - i_1 > 2k$, то при значении `dp` меньше нуля (с учётом жителей в столбцах правее i_1) уже можно заканчивать, а при значении хотя бы ноль его не выгодно использовать при пересчёте.

Итого у нас $m \cdot 3^{n-1}$ состояний, из каждого состояния мы делаем не более $4n \cdot 3^{n-1}$ пересчётов, каждый из которых делается за $O(n)$.