Разбор задачи «Постановочное фото»

Посмотрим на каждый цвет c, найдем самое левое и самое правое вхождение этого цвета, обозначим их как $l[c]$ и $r[c]$. Покажем, что если существует решение, то в нем должно быть действие, которое заполняет цветом c отрезок от $l[c]$ до $r[c]$. Действительно, отрезок меньше быть не может, потому что клетки $l[c]$ и $r[c]$ нужно покрасить, а отрезок больше не нужен, потому что клетки за пределами отрезка от $l[c]$ до $r[c]$ все равно потом придется перекрасить.

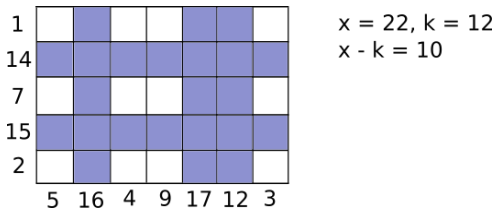
Таким образом, мы получили набор действий, которые должны быть в решении, если оно есть. Осталось сделать их в правильном порядке и проверить, что все получилось. Чтобы получить правильный порядок, можно например отсортировать отрезки по убыванию длины. А чтобы проверить, что последовательность действий приводит к правильному результату, можно использовать дерево отрезков или проход сканирующей прямой по событиям: цвет c и цвет c закончился. Оба подхода работают за $\mathcal{O}(n \log n)$.

Но есть и более простое решение задачи, которое к тому же работает за линейное время. Алгоритм похож на проверку правильности скобочной последовательности. Будем двигаться слева направо и хранить для каждого цвета его текущее состояние. Состояний бывает три: до начала отрезка, внутри отрезка и после конца отрезка. Также будем хранить в стеке цвета, которые находятся во втором состоянии, в том порядке, в котором они начались. Тогда посмотрим на цвет очередной клетки. Если такого цвета еще не было (состояние 1), то добавим его в стек и переведем в состояние 2. Если цвет сейчас в стеке (состояние 2), то вытащим из стека все до него, и переведем все, что вытащили в состояние 3. Если же цвет сейчас уже в состоянии 3, то решения не существует. Суммарное время работы алгоритма $\mathcal{O}(n)$.

Разбор задачи «Беспилотное такси»

В первой подзадаче было достаточно явно хранить все поле в памяти и искать кратчайший путь при помощи обхода в ширину.

Чтобы решать следующие подзадачи, немного изменим модель. Будем записывать в строках и столбцах последний момент времени (а, точнее, индекс запроса), когда строка или столбец была очищена. Изначально все строки и столбцы были очищены в момент времени 0. Пусть нам в момент времени x надо ответить на запрос для такси с проходимостью k. Тогда это такси может ездить по всем строкам и столбцам, в которых записанное число не меньше $x - k$.



В каждом из запросов область допустимых клеток представляет собой «решетку», то есть допустимые клетки являются объединением некоторого количество строк и столбцов.

Кратчайший путь между узлами этой решетки равен $|r_1 - r_2| + |c_1 - c_2|$ и похож на русскую букву «Г».

Кратчайший путь между клетками лежащими на звеньях решетки выглядит следующим образом:

- Если обе клетки лежат в одной группе активных строк или столбцов, то как и в предыдущем случае путь похож на букву «Г».
- Иначе из каждой клетки надо дойти до какого-то из двух ближайших узлов, а потом пройти между этими узлами кратчайшим путем. В данном случае можно показать что оптимальный путь будет похож на русскую букву «П» или на ломаную из трех звеньев, похожих на часть лестницы. Поскольку разбирать все возможные случаи сложно, самый простой способ обработать этот случай — перебрать ближайший узел для каждой из двух точек и взять минимум по всем четырём вариантам пути.

Пути не существует, если либо начальная клетка, либо конечная не проходима, или не выполняется не один из способов выше.

Теперь приступим к решению задачи. Сначала проверим что ответ существует, для этого проверим что изначальные клетки проходимы и одновременно есть хоть одна активна строка и один активный столбец или обе клетки принадлежат одной группе строк или столбцов.

Чтобы проверить что клетки лежат в одной группе строк или столбцов надо проверить что минимум на отрезке строк или столбцов превосходит $x - k$.

Если клетки находятся не в одной группе строк или столбцов, то надо найти ближайшие узлы к этим клеткам, для этого надо уметь находить ближайшие слева и справа активные строки или столбцы.

Чтобы решить вторую подзадачу надо реализовать описанные выше операции за линейные проходы по массивам строк и столбцов. Асимптотика $O((n + m) \cdot q)$.

Чтобы решить третью подзадачу для запросов с $x - k \leqslant 0$ надо сразу выводить сумму разности координат, иначе нас интересует не более q строк и столбцов, которые можно хранить в какой-нибудь стандартной структуре данных, например `std::set`. Асимптотика $O(q^2)$.

Чтобы решить пятую подзадачу надо реализовать минимум на отрезке при помощи какой-нибудь структуры данных, искать ближайших слева и справа не нужно, ибо ответ существует только для клеток находящихся в одной группе строк.

Чтобы решить шестую подзадачу, в которой все k равны, надо все так же реализовать минимум на отрезке, а для поиска ближайших слева и справа можно использовать `std::set`, каждая строка и столбец добавляется в него и активна только для следующих k запросах, поэтому искать ближайших слева и справа можно при помощи `lower_bound`.

Для решения четвертой и седьмой подзадачи надо реализовать спуск по дереву для поиска ближайшего. В зависимости от эффективности решения оно пройдет либо обе подзадачи либо только четвертую.

Разбор задачи «Экспресс 20/19»

Попробуем решить задачу методом динамического программирования. Посчитаем для каждой пары (v, x) булево значение $d[v, x]$: существует ли путь до вершины v длины ровно x. Эти значения можно посчитать следующим образом: переберем вершину v, пусть до нее существует путь длиной x, и есть ребро $v \to u$ длиной w, тогда есть путь до вершины u длиной $x+w$. Этот алгоритм работает за время $O(mD)$, где m — число ребер в графе, а D — максимальное интересное нам значение длины пути. Заметим, что во второй подзадаче m и D небольшие, поэтому это решение будет работать быстро.

Чтобы решить задачу для большого D, нужно заметить, что нам нужно находить путь с длиной, не точно равной заданной, а находящейся в некотором достаточно широком диапазоне. Давайте для начала решим вспомогательную следующую задачу. Дано множество целых неотрицательных чисел $\{a_i\}$, не превышающих D, и некоторая константа α. Мы хотим уметь проверять по заданному r, правда ли среди чисел a_i есть число из диапазона $[r, \alpha \cdot r]$. Заметим следующий факт. Пусть в множестве есть три числа $x < y < z$, и $z \leqslant \alpha \cdot x$, тогда число y можно удалить из множества и ответ на запрос для любого r не изменится, потому что если в запрашиваемом диапазоне было число y, то в нем также было x или z. Действительно, пусть диапазон $[r, \alpha \cdot r]$ содержал число y, но не содержал x и z. Это значит, что $x < r$ и $z > \alpha \cdot r$, а значит $z > \alpha x$. Давайте «проредим» таким образом наше множество $\{a_i\}$, удаляя ненужные элементы пока возможно. Сколько элементов у нас останется? Посмотрим на оставшиеся элементы в отсортированном порядке. Тогда для любого i выполняется $a_{i+2} > \alpha \cdot a_i$ (иначе можно было бы удалить a_{i+1}). Таким образом, $a_n \geqslant \alpha^{\lfloor n/2 \rfloor} a_0$, а значит число элементов $O(\log_\alpha D)$.

Попробуем применить эту идею в нашей задаче. Будем хранить для каждой вершины значение $d[v]$ — «прореженное» множество длин путей до этой вершины в виде отсортированного массива. Тогда общее число значений, которое нам нужно будет хранить, равно $O(n \cdot \log_\alpha D)$, для ограничений задачи это немного. Осталось научиться достаточно быстро вычислять эти значения. Будем делать это почти так же, как в исходном алгоритме. Переберем вершину v, пусть до нее существуют пути с длинами из множества $d[v]$, и есть ребро $v \to u$ длиной w, тогда нужно добавить в множество

длин для вершины u значения $d[v][i] + w$. Это можно сделать за линейное время с помощью алгоритма слияния отсортированных массивов. После слияния, полученный массив нужно «проредить» с помощью процедуры, описанной выше. Это можно также сделать за линейное время с помощью стека.

Таким образом, финальное время работы алгоритма будет $O((n + m) \cdot \log_\alpha D)$.

Разбор задачи «Чёрная дыра»

Заметим, что если на один тот же запрос нам дали один и тот же ответ два раза, этот ответ обязан быть верным. Поэтому для подзадачи 1 ($n \leqslant 1000$, $q \leqslant 30$) мы можем выполнять обычный бинарный поиск, повторяя каждый запрос три раза и считая верным тот ответ, который повторился дважды. Для подзадачи 2 ($n \leqslant 1000$, $q \leqslant 21$) заметим, что третий раз запрос необходимо повторять, только если первые два ответа отличались, и после этого ответы на все запросы обязательно будут верными. Таким образом, количество запросов будет равно $3\lceil \log_2 n \rceil$ и $2\lceil \log_2 n \rceil + 1$ соответственно.

Во всех остальных подзадачах требуется уложиться в минимальное число запросов, достаточное при любой стратегии ответа на запросы при данном n. Первые несколько подзадач ($n \leqslant 12$ или $n \leqslant 25$) можно пройти, реализовав перебор возможных стратегий. В качестве возможных оптимизаций можно задавать состояние перебора мультимножеством всех полученных ответов, а также пользоваться тем, что количество разрешенных запросов невелико (не более 9 для $n \leqslant 25$).

Чтобы получить полиномиальное по времени решение заметим следующее. Пусть в качестве ответов нам сообщили, что ответ принадлежит префиксам длины $p_1 \leqslant p_2 \leqslant \ldots$ и суффиксам длины $s_1 \leqslant s_2 \leqslant \ldots$. Тогда ответ про префикс длины p_2 не мог быть неверным, так как тогда неверным был бы и ответ про префикс длины p_1; аналогично ответ про суффикс длины s_2 также точно является верным. Поэтому состояние поиска можно однозначно задать числами p_1, p_2, s_1, s_2. Будем вычислять величины ans_{p_1,p_2,s_1,s_2} — необходимое количество запросов, чтобы отгадать число в данном состоянии. Число x обязано принадлежать объединению диапазонов $[n-s_1+1, p_2] \cup [n-s_2+1, p_1]$; если длина этого объединения равна 1, то величина равна 0. В противном случае для произвольного запроса `? x` мы переходим в одно из двух состояний, параметры которых легко вычисляются (обозначим эти состояния $L(x)$ и $R(x)$); оптимальный запрос x должен минимизировать $\max(ans_{L(x)}, ans_{R(x)})$. Для вычисления $ans_{\ldots}$ воспользуемся динамическим программированием. В таком решении мы имеем $O(n^4)$ состояний, в каждом из которых возможно $O(n)$ переходов, поэтому суммарная сложность составляет $O(n^5)$. Такое решение набирает 30–35 баллов (дополнительно к 15 баллам за подзадачи 1 и 2).

Рассмотрим несколько способов соптимизировать это решение:

- Пусть префикс длины p_1 и суффикс длины s_1 не пересекаются. Это означает, что к этому моменту один из ответов точно был неверным, и в оставшемся диапазоне возможных значений можно применить обычный бинарный поиск. Перейдем к более удобным обозначениям для случая, когда это неверно: пусть b равно длине пересечения $[1, p_1] \cap [n - s_1 + 1, n]$, $a = p_1 - b$, $c = s_1 - b$.

- Заметим, что значения ans_{p_1,p_2,s_1,s_2} и $ans_{p_1+d,p_2+d,s_1-d,s_2-d}$ совпадают, а стратегии для этих состояний отличаются сдвигом всех запросов на d. Это позволяет задавать состояние числами $p_2 - p_1, p_1 + s_1, p_2 + s_1$, и сложность решения с такой оптимизацией составляет $O(n^4)$ (35–40 баллов).

- Заметим, что в любом состоянии оптимальное количество запросов после ответа `<` на запрос `? x` не уменьшается при увеличении x; аналогично, количество запросов после ответа `>=` x не увеличивается. Это означает, что оптимум $\max(ans_{L(x)}, ans_{R(x)})$ можно искать бинарным поиском по x, что сокращает сложность до $O(n^3 \log n)$ (вместе с предыдущей оптимизацией 40–48 баллов).

- Несложно показать, что, например, при увеличении c позиция оптимального ответа не уменьшается; это позволяет искать оптимальный переход амортизированно за $O(1)$. Вместе с предыдущими оптимизациями получаем сложность $O(n^3)$ (55–60 баллов).

- Заметим, что значение ответа составляет $O(\log n)$. Поменяем местами значение ответа и один из параметров ДП: пусть $maxc_{k,a,b}$ равно максимальному значению c, при котором в состоянии (a, b, c) возможно угадать число за k запросов, либо $-\infty$, если это нельзя сделать ни при каком c. Тогда возможны следующие переходы:

  - если $maxc_{k-1,a,b} \neq -\infty$, то $maxc_{k,a,b} \geqslant maxc_{k-1,a,b} + 2^{k-1} - b$. Действительно, в состоянии $(a, b, c + 2^{k-1} - b)$ сделаем запрос, приводящий к состояниям (a, b, c) и $(b, 0, 2^{k-1} - b)$, во втором из них применим обычный бинпоиск.
  - если $a \geqslant 2^{k-1} - b$, то по аналогичному рассуждению $maxc_{k,a,b} \geqslant maxc_{k-1,a-(2^{k-1}-b),b}$.
  - пусть мы делаем запрос, разбивающий среднюю часть длины b на части длины x и $b - x$, тогда переходы совершаются в состояния $(a, x, b-x)$ и $(x, b-x, c)$. Тогда, если выполняется $maxc_{k-1,a,x} \geqslant b - x$, то $maxc_{k,a,b} \geqslant maxc_{k-1,x,b-x}$.

  Вместе со всеми предыдущими оптимизациями получаем сложность $O(n^2 \log n)$ (60–70 баллов, +15 за первые две подзадачи).

Для того, чтобы пройти оставшиеся подзадачи, требуется найти стратегию локально и аккуратно сохранить её в код программы. Пусть $f(k)$ равно максимальному n, при котором можно угадать число за k запросов. Заметим, что стратегия для $f(k)$ позволяет угадать число за k запросов и для любого меньшего n. Тогда для решения задачи требуется найти стратегии для $f(1), f(2), \ldots, f(maxk = 19)$ и $maxn = 30\,000$.

При конкретном k и n стратегию можно представить в виде решающего дерева глубины k. Такое дерево возможно получить локальными вычислениями на своём компьютере, даже если решение не укладывается по времени в проверяющей системе.

Чтобы дерево не получилось слишком большим при больших k, заметим следующее:

- вершины дерева, соответствующие одинаковым состояниям поиска, можно сохранять только один раз.
- если нам требуется сохранить сразу несколько стратегий, пересекающиеся состояния между этими стратегиями также можно сохранить только один раз.
- пускай в каком-либо состоянии поиска $b = 0$ (т.е. наименьшие префикс и суффикс не пересекаются). Тогда мы можем применить обычный бинпоиск, и такую ветку дерева можно явно не сохранять.

С использованием этим оптимизаций эталонное решение жюри строит сжатое решающее дерево с < 2000 вершин. Построенный код занимает 72 килобайта, а построение занимает 3 минуты и использует <6 гигабайт памяти.

Разбор задачи «Гирлянда»

Так как в первой подзадаче $n \leqslant 15$, то можно перебрать все 2^n подстрок исходной строки, проверить, какие из них соответствуют красивым гирляндам и выбрать максимальную.

Назовем блоком максимальную по длине последовательность подряд идуших шариков в гирлянде. Тогда гирлянда имеет вид $\text{блок}_1, \text{флажок}_1, \text{блок}_2, \text{флажок}_2, \ldots, \text{флажок}_k, \text{блок}_{k+1}$. Назовем длиной i-го блока число шариков в нем. Заметим, что некоторые блоки могут иметь длину ноль. Назовем блочным представлением гирлянды последовательность длин блоков (l_i) для $i = 1..(k+1)$.

Заметим, что удаление флажка из гирлянды приводит к объединению блоков, которые он разделял. Кроме того, гирлянда является красивой тогда и только тогда, когда она состоит из не менее чем двух блоков и все блоки в ней имеют одинаковую длину l. Соответственно, если удалять из гирлянды только шарики, то самая длинная красивая гирлянда, которую возможно получить, будет иметь $k + 1$ блок, длина каждого из которых будет $l = \min l_i$.

Во второй подзадаче исходная гирлянда содержит не более двух флажков. Если флажок один, то его удалять нельзя и в ответе $l = \min(l_1, l_2)$. Если флажков два, то мы можем удалить не более

одного из них, и для каждого из трех полученных вариантов найти максимальную длину красивой гирлянды.

В третьей подзадаче $k \leqslant 15$, поэтому возможно перебрать все 2^k вариантов удаления флажков в блочном представлении и подсчитать для каждого из них ответ за $O(k)$.

Для решения последующих подзадач будем перебирать l в диапазоне от 0 (оставляем только флажки) до $\sum l_i$.

В четвертой подзадаче для каждого l будем решать задачу проходом по гирлянде слева направо. В начале возьмем первые l шариков (возможно, пропуская флажки), потом следующий за ними флажок (возможно, пропустив несколько шариков), затем снова l шариков, и так далее, пока гирлянда не будет исчерпана. Пусть p – число полных блоков длины l которые были получены, тогда самая длинная красивая гирляндой с блоком длины l будет иметь длину $(l+1)(p+1)-1$. Так как для каждого l обработка производится за $O(n)$, то суммарное время работы будет $O(n^2)$.

В пятой подзадаче можно было ускорить решение проходя не по самой гирлянде, а по ее блочному представлению, таким образом обработка каждого l будет производиться за $O(k)$.

Заметим, что решение задачи при фиксированном l можно разбить на следующие операции: переход от шарика с номером i к шарику с номером $i+l-1$, переход от шарику к следующему за ним флажку, переход от флажка к следующему за ним шарику.

Для эффективного выполнения первой операции, предподсчитаем для каждого шарика его номер, пропуская флажки. Так же для каждого номера шарика предподсчитаем его местоположение в гирлянде. Обе этих величины можно посчитать за один линейный проход. Тогда мы сможем за константное время находить номер текущего шарика, прибавить к нему $l-1$ и получить местоположение последнего шарика в блоке.

Для выполнения последних двух операций за константное время предподсчитаем для каждого элемента гирлянды, следующий за ним элемент другого типа. Это возможно сделать линейным проходом, начиная с конца гирлянды.

Производя каждую операцию за $O(1)$ мы получим ответ для заданного l за $O(p)$. Заметив, что $p \leqslant n/l$, получим общее время выполнения всех запросов $O(\sum_1^n n/l)$. Вынеся n, получим префикс гармонического ряда длины n, который растет как $O(\log n)$. Таким образом, общее время работы составит $O(n \log n)$.

Разбор задачи «Классные парты»

Для решения задачи нам потребуется сделать несколько ключевых наблюдений.

Лемма 1. Если отрезок $[L_i, R_i]$ для парты i-го типа полностью вложен в отрезок $[L_j, R_j]$ для парты j-го типа: $L_j \leqslant L_i \leqslant R_i \leqslant R_j$, то в оптимальном решении можно не использовать парты i-го типа.

Доказательство. Заменив парту i-го типа на парту j-го типа, мы не увеличим неудобство для любого сидящего за ней школьника.

Первым шагом удалим все отрезки, которые вложены в другие. Теперь порядки сортировки парт по возрастанию L_i и по возрастанию R_i совпадают.

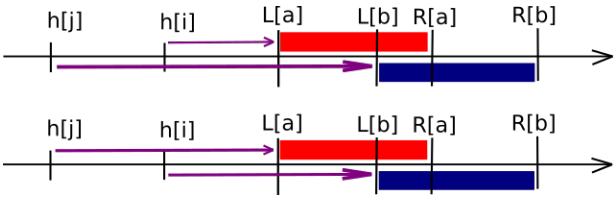
Начнём с решения следующей вспомогательной задачи: пусть есть только одна группа из $2n$ школьников и выбраны типы парт для класса. Требуется разместить школьников за партами, чтобы минимизировать их суммарное неудобство. Оказывается, что для этого оптимально отсортировать школьников по росту, а парты — по левой границе, и жадно разместить школьников за партами в этом порядке.

Лемма 2. Отсортируем школьников по росту, также отсортируем парты по возрастанию левой границы. Оптимально рассадить школьников следующим образом: первых двух школьников надо посадить за первую парту, следующих двух — за вторую, и так далее.

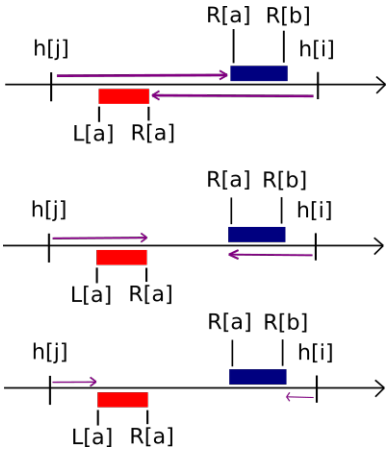
Доказательство. От противного: пусть есть два школьника с ростом $h_i > h_j$, причем школьник i сидит за партой типа a, а школьник j сидит за партой типа b, где $L_a < L_b$ (и, соответственно, $R_a < R_b$). Поменяем их местами и заметим, что их суммарное неудобство не увеличится. Для этого сделаем следующее. Изобразим на прямой отрезки $[L_a, R_a]$, $[L_b, R_b]$, точки h_i и h_j. Проведем вектор из точки, соответствующую росту школьника, до ближайшей точки отрезка парты, за которой он сидит. Суммарное неудобство этих двух школьников равно сумме длин этих векторов.

Рассмотрим три случая.

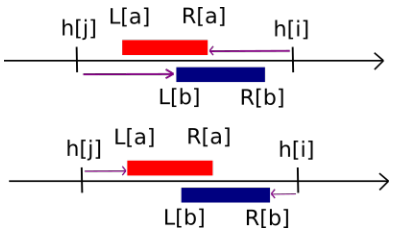
- Оба вектора направлены в одну сторону. В этом случае векторы обязательно имеют общий подотрезок. Поменяем местами концы векторов, что соответствует тому, что мы поменяем местами школьников. Суммарная длина векторов не изменится.



- Векторы направлены в разные стороны, и имеют общий подотрезок (возможно точку). Поменяем местами концы векторов, что соответствует тому, что мы поменяем местами школьников. Суммарная длина векторов уменьшится на удвоенную длину их общего подотрезка. После этого возможна ситуация, что вектор из точки ведет не в ближайшую точку отрезка, но, исправив это, мы лишь уменьшим его длину.



- Векторы направлены в разные стороны и не имеют общих точек. Это возможно лишь в ситуации, когда $h_j \leqslant L_b < R_a \leqslant h_i$. Тогда суммарное неудобство этих двух школьников равно $(L_b - h_j) + (R_a - h_i)$, а если их поменять местами, то будет $\max(0, L_a - h_j) + \max(0, h_i - R_b)$, что не больше.



Пусть мы купили некоторые парты. Отсортируем их по возрастанию L_i. Отсортируем школьников в каждой группе по возрастанию роста. Тогда в каждой группе мы знаем, какие школьники будут сидеть за какой партой. Заметим, что множество школьников, сидящих за t-й по возрастанию левой границы партой не зависит от размера этой парты: это $(2t-1)$-й и $2t$-й по росту школьник в каждом классе. Воспользуемся теперь этим фактом, чтобы выбрать, какие парты требуется купить.

Рассмотрим $(2t-1)$-го и $2t$-го по росту школьника в каждом классе. Объединим их в множество из $2m$ школьников. Все они будут сидеть за одной партой, поэтому надо купить им парту, которая минимизирует их суммарное неудобство.

Решение, которое перебирает все варианты парт для каждого такого множества и минимизирует суммарное неудобство, работает за $O(mnk)$ и проходит подзадачи с 1 по 7, набирая 70 баллов.

Чтобы решить оставшиеся подзадачи потребуется быстрее подбирать оптимальную парту для множества школьников.

Лемма 3. Рассмотрим два массива целых чисел $A = [a_1, a_2, \ldots, a_l]$ и $B = [b_1, b_2, \ldots, b_l]$, причем для всех i выполнено $a_i \leqslant b_i$. Тогда если эти массивы отсортировать по возрастанию, то по прежнему для всех i будет выполняться $a_i \leqslant b_i$.

Доказательство. Пусть число a_i является k-м по возрастанию среди чисел массива A. Тогда есть хотя бы k чисел в массиве b, меньших или равных a_i: b_i и все b_j для таких j, что $a_j \leqslant a_i$. Значит k-е по возрастанию число массива B не превышает a_i, что и требовалось.

Лемма 4. Рассмотрим два множества школьников. Пусть в первом множестве школьники имеют рост $[h_1, h_2, \ldots, h_{2m}]$, а во втором $[g_1, g_2, \ldots, g_{2m}]$, причем для всех i выполнено $h_i \leqslant g_i$. Пусть для первого множества оптимальна парта, подходящая для отрезка $[L_a, R_a]$, а для второго парта, подходящая для отрезка $[L_b, R_b]$. Тогда $L_a \leqslant L_b$ (и, соответственно, $L_b \leqslant R_b$).

Доказательство. Предположим противное, пусть $L_a > L_b$. Рассмотрим пары школьников вида $h_i - g_i$, заметим, что по доказательству леммы 2, если обменять местами школьников, пересадив школьника с ростом h_i за парту типа b, а школьника с ростом g_i за парту типа a, то суммарное неудобство этих двух школьников не увеличится. Проделаем такую операцию с каждой парой школьников. Суммарное неудобство всех школьников не увеличится, значит хотя бы для одного из множеств $[h_1, h_2, \ldots, h_{2m}]$ и $[g_1, g_2, \ldots, g_{2m}]$ суммарное неудобство не увеличится. Значит можно либо заменить одному из этих множеств парту, сделав $a = b$, либо вообще оптимальнее поменять назначения парт местами, улучшив оба суммарных неудобства. Противоречие.

Рассматривая леммы 3 и 4 в совокупности, получаем, что переходя от школьников на позициях $2t-1$ и $2t$ в отсортированных по возрастанию роста группах к школьникам на позициях $2(t+1)-1$ и $2(t+1)$, номер подходящей для них парты в отсортированном порядке парт не уменьшается.

Рассмотрим школьников на позициях $2t-1$ и $2t$ для $t = n/2$. Подберём для них оптимальную парту за $O(mk)$. Пусть она имеет тип k_t. Заметим, что теперь для школьников на меньших позициях подходят только парты с номерами не больше, чем k_t, а для школьников на бóльших позициях — с номерами не меньше, чем k_t. Рекурсивно решим задачу для половин, групп на этот раз перебирая только подходящие парты. Время работы получившегося решения $O(mk \log n)$ и оно проходит также подзадачу 9.

Чтобы пройти оставшиеся подзадачи, надо научиться подбирать парту для множества школьников оптимальнее, чем пробуя все варианты за время $O(mk)$. Для этого отсортируем всех школьников в множестве, после этого, зная параметры L_a и R_a для парты, мы можем двоичным поиском определить, какому отрезку школьников она подходит по росту, а суммарное неудобство остальных школьников можно посчитать за $O(1)$ с использованием префиксных сумм. Получаем время работы $O((m + n \log n + k) \log n)$.

Разбор задачи «Робогольф»

Чтобы решить первую подзадачу, можно воспользоваться методом динамического программирования: $dp_{i,j,player}$ — фишка сейчас находится в клетке с координатами i, j и сейчас ходит игрок $player$. Обозначим за $player = 1$ первого игрока, который минимизирует, а за $player = 2$ игрока, который максимизирует цену игры. Тогда ответ на задачу это сумма $dp_{i,j,1}$ по всем $1 \leqslant i \leqslant n$ и $1 \leqslant j \leqslant m$.

Если в клетке изначально стоит значение, то оба значения в динамике равны ему, иначе получаем следующие переходы:

$$dp_{i,j,1} = \min\left(dp_{i+1,j,2}, dp_{i,j+1,2}\right)$$

$$dp_{i,j,2} = \max\left(dp_{i+1,j,1}, dp_{i,j+1,1}\right)$$

Теперь избавимся от третьего параметра в решении с динамическим программированием. Для этого, покрасим доску в шахматную раскраску, и переберем цвет, на котором будет делать ходы первый игрок. Тогда, если ход в клетке (i, j) делает первый игрок, и в клетках $(i+1, j)$ и $(i, j+1)$ нет ловушек, $d_{i,j} = \min(\max(d_{i+2,j}, d_{i+1,j+1}), \max(d_{i+1,j+1}, d_{i,j+2})) = \max(d_{i+1,j+1}, \min(d_{i+2,j}, d_{i,j+2}))$.

Будем рассматривать диагонали с одинаковой суммой $i+j$, и будем идти по диагоналям по убыванию суммы. Тогда, $d_{i,j} = d_{i+1,j+1}$, если в клетках $(i+d_i, j+d_j)$ нет ловушек ($d_i \leqslant 2$, $d_j \leqslant 2$, $d_i + d_j \leqslant 2$), и $d_{i+1,j+1} \geqslant d_{i+2,j}$ или $d_{i+1,j+1} \geqslant d_{i,j+2}$. Заметим, что второе условие означает, что $d_{i+1,j+1}$ не является локальным минимумом в массиве значений на диагонали. Но локальный минимум мог появиться только из-за ловушки, и уже через одну диагональ он исчезнет. Поэтому, число в клетке (i, j) нужно пересчитывать, если в клетке $(i + d_i, j + d_j)$ находится ловушка ($d_i, d_j \leqslant 4$). Назовём такие клетки *интересными*.

Будем поддерживать массив чисел, которые стоят на диагонали. При переходе от одной диагонали к следующей, пересчитаем значения в интересных клетках. Также, будем поддерживать сумму на диагонали, которую будем пересчитывать при каждом изменении массива. Сложим эти суммы по всем диагоналям. Такое решение работает за $O(n + m + k)$.

Рассмотрим две диагонали, содержащими интересные клетки, между которыми диагоналей с интересными клетками нет. Значения динамического программирования между этими диагоналями не будут изменяться. Поэтому, можно пропускать диагонали, на которых нет интересных клеток. Теперь вместо массива нужно использовать хеш таблицу или дерево поиска. Такое решение работает за $O(k)$ или за $O(k \cdot \log k)$.

Разбор задачи «Поиск идеи»

В первых двух подзадачах есть возможность разжать строку t в явное представление, и решить стандартную задачу поиска подстроки в строке. Начиная с третьей, вся строка не умещается в разумный размер. Как следствие, необходимо придумать некоторое компактное представление, позволяющее обрабатывать не очень много блоков данных, каждый из которых получится обработать достаточно быстро.

В третьей подзадаче можно заметить, что строка t является повторенной некоторое количество раз строкой, которую записали в первом блоке. Для поиска подстроки в такой строке необходимо повторить период столько раз, чтобы он стал длиннее удвоенной строки p. Количество вхождений в полученную строку можно посчитать за линейное от ее длины время. Несложно заметить, что добавление каждой следующей копии будет увеличивать количество вхождений на одинаковое значение. Вычислить это значение можно увеличив количество повторений на 1 и еще раз посчитав количество вхождений.

В четвертой подзадаче полученная строка состоит из не очень большого (порядка числа операций) количества блоков из одинаковых букв. Можно заметить, что если блок существенно длиннее t, то его можно укоротить до удвоенной длины t, и это в большинстве случаев не поменяет количества вхождений. Единственный случай когда это не так — строка t состоящая только из этой буквы. Но в этом случае понять насколько уменьшилось количество вхождений также не сложно — оно равно количеству удаленных букв. После удаления, длина строки не превосходит длины паттерна в квадрате, и ее можно обработать так же как в первых подзадачах.

В 5 и 6 подзадаче после того, как длина строки стала больше, чем 10^7, дальше будут копироваться только ее префиксы. Заметим, что в тот момент, когда длина строки стала больше, чем 10^7, ее длина не больше $2 \cdot 10^7$. Будем хранить наибольший суффикс, являющийся префиксом паттерна, аналогично тому, как это делается в алгоритме Кнута-Морриса-Пратта. Если дописываемый префикс короче паттерна, то просто допишем его по одному символу, пересчитывая наибольший суффикс с помощью заранее насчитанной префикс-функции паттерна. Иначе, допишем только его префикс длины равной длине паттерна. Все остальные вхождения находятся внутри префикса, и их число можно найти с помощью бинарного поиска или префиксных сумм на заранее посчитанных вхождениях в строку из которой копируются префиксы. Наибольший суффикс в этом случае так же равен наибольшему суффиксу этого префикса, то есть его префикс-функции.

В 7 и 8 подзадаче копироваться могут только существующие до начала копирования символы. Будем хранить строку как последовательность подотрезков строк написанных операциями типа 1. Количество таких подотрезков после каждой операции увеличится не более чем в два раза. Так как операций не более 20, общее количество подотрезков будет не более 2^{19}. После этого можно склеивать эти подотрезки способом аналогичным 5 и 6 подзадаче.

Во всех следующих подзадачах нет ограничений на копирования и количество операций копи-

рования достаточно велико для предыдущего решения. Такие копирования можно осуществлять с помощью персистентных структур данных, например декартового дерева или другого сбалансированного дерева поиска.

Для того, чтобы скопировать отрезок необходимо вырезать его из старой версии дерева. При этом так как мы используем пресистентную структуру данных, старая версия остается корректной, и можно объединить вырезанный подотрезок со старой версией. К сожалению, части отрезка который необходимо скопировать может еще не быть. В таком случае дописанная часть состоит из большого количества повторений суффикса к которым дописали какую-то подстроку. Строку записанную как персистентное дерево можно объединить с самой собой, что позволяет воспользоваться алгоритмом аналогичным бинарному возведению в степень.

В результате такого процесса строка получается записанной в виде ориентированного ациклического графа, причем у каждой вершины есть «левое» и «правое» ребро, соотвествующие левому и правому сыну в дереве поиска. Каждой вершине этого графа соответствует некоторая строка, равная конкатенации строки, соответствующей ее левому ребру, символа записанного в вершине, и строки соответствующей правому ребру. Если для каждой вершины хранить длину наибольшего суффикса, являющегося префиксом паттерна, и наибольшего префикса, являющегося суффиксом паттерна, то можно будет посчитать количество вхождений на стыке, и если прибавить к нему количество вхождений в левой и правой половине, то получится количество вхождений в строку, соответствующую этой вершине. Этого достаточно для прохождения 9, 10 и 11 подзадач. Суммарная сложность такого алгоритма $\mathcal{O}(n \log T \log L + T \cdot m)$, где T — количество вершин в получившемся персистентном дереве поиска, которое порядка $\mathcal{O}(n \log^2 L)$.

К сожалению, быстро пересчитывать длину наибольшего суффикса, являющегося префиксом паттерна, и наибольшего префикса, являющегося суффиксом паттернана, не получается, поэтому для дальнейшего ускорения применим другой подход. Будем хранить не наибольший префикс, являющийся суффиксом паттерна, а наибольшей префикс являющийся подстрокой паттерна, и аналогично, наибольший суффикс являющийся подстрокой паттерна. Такие префикс и суффикс могут быть только длиннее, поэтому никаких вхождений потеряно не будет.

Теперь необходимо научиться решать две строковые задачи:

- Для подсчета количества запросов на склейке: даны две подстроки паттерна, необходимо найти количество вхождений паттерна в их конкатенации.
- Для пересчета префикса и суффикса: даны две подстроки паттерна, какой наибольший префикс второй можно приписать к первой, чтобы она осталась подстрокой.

Для решения первой задачи рассмотрим структуру данных «дерево префикс-функции». Если каждый префикс строки считать вершиной графа, и провести ребро из префикса в его префикс-функцию, то этот граф образует дерево, корнем которого является пустая строка. Заметим, что чтобы найти вхождение, нужно найти суффикс первой строки и префикс второй, которые являются префиксом и суффиксом паттерна соотвественно, и при этом их суммарная длина равна длине патерна. Множество таких суффиксов является путем до корня от некоторой вершины в дереве префикс-функции для паттерна, а префиксов — путем до корня от некоторой вершины в дереве префикс-функции для перевернутого паттерна. Эти вершины можно найти с помощью двоичных подъемов. Таким образом эта задача свелась к поиску пересечения двух путей до корня в двух деревьях. Эту задачу можно решить в offline для всех запросов сразу. Для этого в одном дереве нужно сделать эйлеров обход и построить дерево отрезков на получившемся массиве, а во втором дереве сделать обход в глубину, который считает количество вершин на пути до текущей для всех вершин первого дерева с помощью построенного дерева отрезков. Сложность такого решения будет $\mathcal{O}((T + |p|) \log |p|)$.

Для решения второй задачи необходимо построить для паттерна суффиксный массив и структуру данных для подсчета наибольшего общего префикса двух подстрок за $\mathcal{O}(1)$ с помощью разреженных таблиц. Подстроки, начинающиеся с данной, представляют собой подотрезок суффиксного массива. При этом этот подотрезок можно интерпретировать как суффиксный массив для суффикса, начинающегося с конца этой подстроки. Его можно найти с помощью бинарного поиска и запросов наибольшего общего префикса.

Максимальный префикс второй строки, который можно приписать к первой, можно искать с помощью еще одного бинарного поиска. Для этого надо в подотрезке суффиксного массива найти строку, ближайшую к конкатенации этих строк, и взять одну из двух соседних с ней. Сравнивать строки можно за $\mathcal{O}(1)$, так как они являются подстроками паттерна. Сложность этой части решения будет $\mathcal{O}(|p| \log |p| + T \log |p|)$.

Таким образом, суммарное время работы алгоритма $\mathcal{O}(n \log^2 L \log |p|)$, чего должно быть достаточно для прохождения 12 подзадачи, и возможно 13 при аккуратной реализации.

Для прохождения последней подзадачи может потребоваться соптимизировать число T уменьшив его до $\mathcal{O}(n \log L)$. Для этого необходимо выбрать хорошее дерево поиска, которое будет всегда небольшой высоты, его будет легко удваивать, не создавая большого числа дополнительных вершин, а так же, которое будет позволять объединять много вершин, получающихся при бинарном возведении в степень, за логарифмическое время, создавая при этом логарифмическое число дополнительных вершин. Примером такого дерева поиска является 2-3-дерево. Оно всегда имеет высоту не более $\log T$, позволяет удваиваться просто добавляя новый корень с двумя одинаковыми сыновьями, и умеет объединяться за разность высот, что позволяет выполнять нужную операцию максимально эффективно. Вероятно, существуют и другие способы оптимизации количества вершин. Таким образом, суммарное время работы алгоритма $\mathcal{O}(n \log L \log |p|)$.