Разбор задачи «Любитель камней»

Автор задачи: Александр Шеховцов, разработчик задачи: Тимофей Ижицкий

В первой подгруппе можно предпосчитать для каждой подпоследовательности ее счет, и отвечать с помощью этого на запрос. Во второй подгруппе можно было написать динамику по строке s и t (примерно как в алгоритме поиска наибольшей общей подпоследовательности), но это не очень разумное решение, поэтому оно подробно не описывается.

В третьей подгруппе, ответ зависит только от длины строк — $\lfloor \frac{|s|-1}{|t|-1} \rfloor$.

Когда в задаче стоит вопрос о том, что нужно максимизировать какой-то минимум, часто приходит идея о бинарном поиске по ответу, давайте попробуем его сделать. Будем делать бинарный поиск по условию «правда ли, что есть подпоследовательность, что расстояние между соседними индексами хотя бы k» (где k — это значение, которое мы проверяем в бинарном поиске). Тогда можно такие индексы набирать жадно, будем идти жадно по индексам s и поочередно набирать текущий символ в t, каждый раз брать минимальный индекс хотя бы на k больше. Таким образом получили решение за $O(n \cdot \log T)$ на запрос, которое проходит 4 и 5 подгруппу.

Чтобы сдать задачу на 100 баллов нужно оптимизировать этот жадник, чтобы он быстрее работал на запрос. Поймем, что по сути в жаднике нужно находить для какой-то позиции pos в строке s, первую позицию $\geqslant pos$, что она равна очередному символу t_i. Давайте для каждой позиции и каждого символа из s это предпосчитаем, тогда в запросе также будем делать бинарный поиск и жадник, только двигаться по s с помощью предпосчитанного массива. Получили решение за $O(n \cdot \Sigma + T \cdot \log T)$, где Σ — размер алфавита (в данном случае 26).

Разбор задачи «Отличительные черты»

Автор задачи и разработчик: Тихон Евтеев

Для начала пройдемся по первым подгруппам и опишем решения, которые могли бы пройти их ограничения:

Подгруппа 1 $(n, m, q \leqslant 500)$

Такие ограничения позволяют для каждого запроса (l, r, p) пройтись по всем отличительным чертам $(1, \ldots, m)$ и проверить, что очередное из них есть у p-го смартфона, и нет ни у какого другого на отрезке за $O(r - l)$

Полученная асимптотика $O(q \cdot m \cdot n)$

Подгруппа 2 $(q \leqslant 5000, s \leqslant 10000)$

Для ответа на запрос (l, r, p) заведем массив длины m, который для каждой черты будет насчитывать правда ли, что она есть только у p. Изначально пройдемся по чертам p и проставим $true$ только в позиции с их номерами.

Затем пройдемся по всем смартфонам на отрезке $[l, r]$, кроме p, и пометим $false$ в массиве ответа. Этот шаг будет работать за $O(s)$ для каждого запроса.

Полученная асимптотика $O(q \cdot (s + m + n)) = O(q \cdot s)$

Сведение 1

Научимся для каждого смартфона и для каждой его черты суммарно за $O(n + m + s)$ находить ближайший слева смартфон с такой же чертой (или -1, если его нет).

Для этого заведем массив $prev$ длины m, изначально заполненный -1. Пройдем слева направо по смартфонам, для каждой черты очередного смартфона i в массиве $prev$ будет лежать ответ. Сохраним его, а на его место положим i, так как это теперь последний смартфон с такой чертой.

Аналогичным образом для каждой черты каждого смартфона найдем ближайший справа смартфон с такой же чертой (или n, еслм его нет), пройдясь справа налево.

Таким образом теперь вместо черт у смартфона i есть несколько пар (l_i^j, r_i^j), и для ответа на запрос (l, r, p) необходимо посчитать количесто таких j, что $l < l_p^j$ и $r_p^j < r$

Подгруппа 3 $q, s \leqslant 100'000, m \leqslant 500$

Применив сведение, описанное выше, для ответа на каждый запрос пройдемся по не более чем m отрезкам (l_p^j, r_p^j) и проверим, что $l < l_p^j$ и $r_p^j < r$.

Полученная асимптотика $O(s + q \cdot m)$

Подгруппа 4 ($p = l$)

В ограничениях подгруппы достаточно проверять только условие $r_p^j < r$. Так что заранее отсортируем набор $\{r_i^j\}$ для каждого i, и для ответа на запрос найдем количество меньших r бинпоиском.

Полученная асимптотика $O((s + q) \cdot \log(m))$

Подгруппа 5 ($l = 1$)

Для каждого смартфона оставим только те пары (l_i^j, r_i^j), для которых $l_i^j = -1$, то есть отсеим те черты, котовые есть хоть у одного смартфона слева, тем самым сведя задачу к предыдущей.

Полезный прием

Научимся двигать границы запроса (l, r, p) в разные стороны, при этом поддерживая ответы для всех элементов на отрезке. Для этого сохраним массив ответом ans длины n , массив подсчета черт cnt длины m и массив xor-ов индексов всех смартфонов с такой чертой $xors$ длины m.

Чтобы подвинуть, например, правую границу вправо, пройдем по всем чертам добавленного смартфона, обновим значение в cnt. Если оно стало 1 или перестало быть 1, обновим ans на позиции, ктоторую можно получить из $xors$, так как всего у одного элемента сейчас есть такое свойство. Еще обновим массив $xors$.

Аналогичным образом можно двигать все границы во все стороны.

Подгруппа 6 ($l_i \leqslant l_{i+1}, r_i \leqslant r_{i+1}$)

Так как границы запросов неубывают, предыдущая техника дает решение за $O(q + s)$

Подгруппа 7 ($q, s \leqslant 100'000$)

Раз уж мы научились двигать границы в разные стороны, можно применить технику МО. Однако стоит быть аккуратным, и считать, что смартфон с m_i уникальными чертами за $m_i + 1$ элемент при разделении запросов на блоки.

Полученная асимптотика $O((s + q)^{(3/2)})$

Сведение 2

Разобьем все запросы по p. Теперь для каждого p у нас есть несколько отрезков из сведения 1: (l_p^j, r_p^j), и несколько запросов (lq_p^i, rq_p^i). Для каждого запроса нужно посчитать количество отрезков, для которых $l_p^j < lq_p^i$ и $rq_p^i < r_p^j$. Если воспринимать l и r как координаты x и y на плоскости, получим задачу количества точек в прямоугольнике.

Подгруппы 8 и 9 ($q, s \leqslant 200'000, 500'000$)

Стандартную задачу количества точек в прямоугольнике, полученную после сведения 2 можно решить за $O((m_p + q_p)^{(3/2)})$ или $O((m_p + q_p) \cdot \log^2(m_p + q_p))$ или $O((m_p + q_p) \cdot \log(m_p + q_p))$ для каждого смартфона p с m_p чертами и q_p запросами с таким p. Для этого можно использовать корневую, двумерное дерево Фенвика или сканлайн с деревом отрезков или деревом Фенвика соответственно, получив разные баллы в зависимости от качества реализации.

Таким образом лучшая полученная асимптотика: $O((s + q) \cdot \log(s + q))$

Разбор задачи «Освещение дорог»

Автор задачи: Тимофей Ижицкий, разработчик задачи: Алексей Михненко

Рассмотрим следующий жадный алгоритм поиска максимального паросочетания в дереве:

- Подвесим дерево за вершину 1 и запустим от неё `dfs`.
- В функции `dfs` сначала рекурсивно вызовем `dfs` от каждого её ребёнка u_i, который найдёт максимальное паросочетание в поддереве вершины u_i. Обозначим размер этого паросочетания за m_{u_i}.
- Пусть c_v равно количеству детей вершины v, которые не были заняты паросочетанием.
- Если $c_v = 0$, то не добавляем новые рёбра в паросочетание. Таким образом, размер паросочетания в поддереве вершины v равен $m_v = \sum m_{u_i}$.

- Если $c_v > 0$, то мы можем добавить одно дополнительное ребро в паросочетание. Добавим в паросочетание рёбро между v и любым его ребёнком, не занятым паросочетанием. В этом случае $m_v = 1 + \sum m_{u_i}$.

Корректность данного алгоритма можно доказать по индукции:

- Докажем, что `dfs`(v) находит максимальное паросочетание в поддереве вершины v. Более того, если существует максимальное паросочетание, в котором вершина v не занята, то он найдёт именно подобное паросочетание.
- База: если v — лист, то корректность алгоритма очевидна.
- Переход: если v не лист, то нетрудно видеть, что $m_v \leqslant 1 + \sum m_{u_i}$. Если все вершины u_i оказались насыщены паросочетанием, то $m_v = \sum m_{u_i}$, так как если мы возьмём любое ребро из v в ребёнка, то размер максимального паросочетания в поддереве этого ребёнка уменьшится на 1, так как иначе было нарушено свойство, что среди максимальных паросочетаний алгоритм находит то, в котором верхняя вершина не занята. В этом случае максимальное паросочетание уже найдено и вершина v им не насыщена, поэтому достаточно ничего не делать.

  Иначе мы можем добавить ребро в любого ненасыщенного ребёнка и увеличить максимальное паросочетание на 1. В этом случае вершина v обязана быть насыщенной, так как без вершины v размер максимального паросочетания равен $\sum m_{u_i}$. Таким образом, данный алгоритм действительно корректен.

В задаче требовалось «включать» и «выключать» рёбра в дереве и находить размер максимального паросочетания в связной по включённым рёбрам компоненте связности. Мы будем оптимизировать описанный выше алгоритм. Сначала запустим его явно и для всех вершин v найдём m_v и c_v, определённые выше. После запросов дерево может разбиться на независимые компоненты связности. В каждой такой компоненте связности выделим самую высокую (наиболее близкую к корню) вершину. Представим, что мы запустили жадный алгоритм в каждой компоненте связности от каждой выделенной вершины, который посчитает значения m_v и c_v. Далее мы будем поддерживать такие значения эффективно.

Разберёмся как меняются m_v и c_v при каждом из запросов:

1. Если ребро (v, u) «выключается», то без потери общности пусть v — предок вершины u. Тогда в поддереве вершины u ничего не изменилось. Вычтем m_u из всех m_p, где p — предок вершины v в новой компоненте связности вершины v. Теперь рассмотрим два случая:

   - Если жадный алгоритм может построить паросочетание, в которое не входит ребро (v, u), то в новой компоненте вершины v никакие значения больше не изменились. Чтобы проверить, можно ли построить такое паросочетание, достаточно проверить, что $c_u > 0$ или $c_v \neq 1$.
   - Иначе, паросочетание в компоненте вершины v уменьшилось ещё на 1. То есть надо вычесть ещё 1 из всех m_p. В этом случае так же надо обновить значения c_p. Так как $c_v = 1 \implies u$ был единственным не насыщенным ребёнком до этого, поэтому надо вычесть 1 из c_v и добавить 1 к m_v. Теперь посмотрим на предка x вершины v (если его не существует или он лежит не в той же компоненте, что и вершина v, то ничего больше делать не надо). До этого мы считали в предке, что вершина v насыщена, а теперь это не так, поэтому надо увеличить c_x на 1. Если после этого $c_x = 1$, то это значит, что до этого вершина x не была соединена с ребёнком. В этом случае надо увеличить m_x на 1 и проделать для предка вершины x такой же алгоритм, как и для вершины v, так как для него получается ровно такой же случай: ребро в его ненасыщенного ребёнка больше нельзя использовать, поэтому это аналогично удалению такого ребра.

2. Если ребро (v, u) «включается», то пусть опять v — предок вершины u. В поддереве вершины u опять ничего не изменилось. Теперь ко всем m_p надо добавить m_u. Теперь заметим, что рассмотренный выше случай для предка x вершины полностью аналогичен добавлению нового ребёнка к вершине x. Поэтому здесь нужно применить такой же алгоритм, но с пропущенным первым шагом.

3. Чтобы найти максимальное паросочетание в компоненте связности вершины v достаточно подняться до самой высокой вершины r в этой компоненте и вывести m_r.

Данный способ наивно можно реализовать за $\mathcal{O}$(глубина вершины v) на запрос. Например в 4-й подгруппе глубина дерева равна $\mathcal{O}(\log n)$, поэтому такое решение будет работать за $\mathcal{O}(n \log n)$.

Чтобы реализовать данный способ в общем случае будем хранить значения m_v и c_v в Heavy-light декомпозиции. Тогда рассмотрим, что надо сделать при «выключении» ребра (v, u). Сначала надо вычесть m_u из всех m_p на каком-то вертикальном пути, что является тривиальным массовым запросом. Теперь надо от вершины v вверх по значениям c_i найти самую длинную последовательность $1, 0, 1, 0, \ldots$ На данном вертикальном пути надо заменить значения c_i на $0, 1, 0, 1, \ldots$. Так же на этом же пути надо вычесть 1 из каждого второго значения m_i.

В случае «включения» ребра, надо по значениям c найти максимальную последовательность $0, 1, 0, 1, \ldots$ и сделать аналогичные предыдущему случаю преобразования.

Чтобы искать наибольшую последовательность вида $1, 0, 1, 0, \ldots$ в HLD можно, например, для каждой чётности сохранить минимальное и максимальное значение. Тогда это можно релизовать обычным спуском по HLD. Чтобы после этого заменить эти значения на $0, 1, 0, 1, \ldots$ достаточно ко всем чётным индексам прибавить 1, а из всех нечётных вычесть. Это так же поддерживается в обычном дереве отрезков.

Таким образом, такое решение можно релизовать за $\mathcal{O}(n \log^2 n)$, что является достаточно эффективным.

В данной задаче так же существует решение за $\mathcal{O}(n \log n)$, использующее link-cut tree. Более того, оно позволяет решать задачу ещё в более общем случае, когда исходное дерево не дано и рёбра могут удаляться и добавляться произвольным образом так, чтобы не образовывалось циклов. Данное решение остаётся читателю в качестве упражнения.

Разбор задачи «Перекрашивание таблицы»

Автор задачи и разработчик: Никита Голиков

Для начала, заметим, что значение $i + j$ не превышает $2n - 2$. Поэтому, если мы берем значение $k \geqslant 2n - 1$, то при взятии по модулю значение в клетке не изменяется. Таким образом, достаточно рассматривать только значения $k \leqslant 2n - 1$.

Подзадача 1

В первой подзадаче были ограничения $n, t \leqslant 100$, поэтому ее можно было решать наивно. Для этого, переберем все значения k до $2n-1$, для каждого за $O(t)$ найдем цвет каждой из покрашенных клеток и посчитаем ответ. Получаем решение за $O(nt)$.

Подзадача 2

Для решения второй подзадачи научимся быстрее считать ответ для фиксированного значения k. Для этого, давайте заранее для каждой диагонали сохраним массив всех цветов покрашенных клеток на этой диагонали, и его отсортируем. Теперь, чтобы найти ответ для фиксированного k, надо перебрать диагональ, найти ее цвет после взятия по модулю k, и найти количество изначально покрашенных в такой цвет клеток. Это можно сделать с помощью двоичного поиска по предподсчитанному массиву. Получаем решение за $O(n^2 \log n)$.

Подзадача 3

Для более быстрого решения задачи, надо научиться по покрашенной клетке (r_i, c_i, v_i) понять, для каких значений k она окажется правильно покрашенной.

Для этого должно выполняться условие $(r_i + c_i) = v_i \pmod{k}$. Это эквивалентно следующим условиям:

1. $v_i < k$
2. $r_i + c_i - v_i$ делится на k.

Давайте для каждого значения k найдем количество клеток, которые окажутся правильно покрашенными, и в конце найдем максимальное значение. Для этого, перебрав клетку (r_i, c_i, v_i), мы хотим прибавить 1 к тем значениям k, для которых значение окажется правильным. Рассмотрим несколько случаев.

1. $r_i + c_i - v_i < 0$ — ни для какого k такая клетка не окажется правильно покрашенной
2. $r_i + c_i - v_i = 0$ — для всех значений $k > v_i$ клетка будет правильно покрашена
3. $r_i + c_i - v_i > 0$ — переберем k - делитель $r_i + c_i - v_i$, и если $k > v_i$, то прибавим 1 к значению.

Для того, чтобы прибавить 1 для всех значений $k > v_i$, прибавим в значении $v_i + 1$, и затем насчитаем префиксные суммы. Если перебрать все делители за время $O(\sqrt{r_i + c_i - v_i})$, то получаем решение за время $O(t\sqrt{n})$ для этой подзадачи.

Подзадача 4

Для полного решения задачи, научимся более быстро перебирать все делители. Для этого, давайте заранее для каждого значения $r_i + c_i - v_i$ сохраним все значения v_i.

Теперь, переберем k, и решим задачу для заданного k. Переберем все значения, кратные k циклом, и для каждого переберем все предподсчитанные значения v_i, и посчитаем количество покрашенных клеток.

Полученное решение работает за $O(n \log n + t \cdot C)$, где C — максимальное ограничение на количество делителей у числа до $2n - 1$. Для максимальных ограничений задачи, C не превышает 288, поэтому решение работает быстро.

Разбор задачи «Игровая зависимость Егора»

Автор задачи и разработчик: Андрей Павлов

Формально в задаче хорошей подмассивом длины m называлась перестановка чисел от 1 до m. Требовалось искать для позиции i и числа x самый маленький подмассив длины хотя бы x и количество таких подмассивов.

Для решения на 7 баллов переберем каждый подмассив, параллельно поддерживая например структуру set, чтобы считать что все числа в подмассиве различны, а тогда нужно чтобы максимум подмассива был равен его размеру. После чего для ответа на запрос пройдемся по всем хорошим подмассивам из позиции и ответим на запрос. Чтобы получить еще 8 баллов заметим, что числа удовлетворяют ограничениям $1 \leqslant a_i \leqslant n$ и будем хранить количество вхождений каждого числа в текущий подмассив, тогда если мы добавляем новый элемент в подмассив и в нем числа перестали быть различными, то после новых добавлений массив хорошим не станет. Таким образом получили решение за $O(n(n + q))$.

Давайте как-нибудь оценим количество хороших подмассивов. Для начала их не может быть больше чем $O(n^2)$, то есть подмассивов всего. Теперь давайте заметим, что подмассив является хорошим если выполняются два свойства:

- Числа в нем различны

- Максимум равняется длине

Давайте зафиксируем число a_i и рассмотрим на каких отрезках он является максимумом, для этого найдем ближайший слева и справа от него больший или равный элемент (если такого нет, то укажем за границы массива). Пусть это l_i, r_i, тогда для всех отрезков таких, что левый конец лежит в полуинтервале $(l_i, i]$, а другой в $[i, r_i)$ максимум на этом отрезке будет равен a_i. Тогда чтобы второе свойство на этом массиве выполнялось, нужно чтобы длина отрезка была a_i, тогда с таким максимумом подмассивов не более чем $\min(i - l_i, r_i - i)$, так как мы можем пойти с меньшего конца и тогда каждая позиция может быть концом не более чем одного хорошего подмассива для данного a_i. А это уже дает оценку в количество хороших подмассивов $O(n \log n)$. Чтобы понять почему это так, давайте посмотрим для элемента a_i в скольких отрезках мы его учтем. Заметим, что когда мы проверяем какой-то элемент, это значит что он был в меньшем из отрезков. Но тогда следующий раз мы его рассмотрим уже в позиции l_i или r_i, то размер отрезка в котором содержится элемент увеличится хотя бы вдвое, потому что если $x < y$, то $x + y > 2x$, а таких увеличений не может произойти суммарно больше чем количество элементов в общем, то есть их не более чем $\log n$, а значит каждый элемент мы учтем не более $\log n$ раз.

Но данную оценку можно улучшить! Вспомним, что в хорошем подмассиве обязательно должна быть единица. Тогда давайте рассмотрим все единицы, они разбивают массив на блоки. Понятно, что хорошие подмассивы должны начинаться в одном блоке и заканчиваться в соседнем, иначе единица встречается либо 0, либо более одного раза, что не удовлетворяет нужному условию хорошего подмассива. Тогда давайте рассмотрим два соседних блока. Хороший подмассив должен быть суффиксом первого блока и префиксом второго блока, но тогда пусть максимум в суффиксе первого блока равен x, тогда размер хорошего подмассива должен быть x, то есть для него существует не более одного префикса из второго блока, причем такого, что максимум на этом префиксе не больше x. Аналогично зафиксируем максимум на префиксе во втором блоке, для него найдется не более одного нужного суффикса. А это значит, что суммарно хороших подмассивов не больше чем размеры этих блоков. Но так как мы рассматриваем только соседние блоки, то суммарное количество хороших подмассивов $O(2n) = O(n)$.

Теперь нетяжело вывести из данной оценки алгоритм поиска всех таких перестановок. Разобьем массив с помощью единиц на блоки, теперь переберем пару соседних блоков и будем перебирать префикс во втором блоке и указателем поддерживать максимальный суффикс, который мы можем взять, чтобы все элементы на текущем префиксе и этом суффиксе были различны, а также чтобы максимум на суффиксе не превосходил текущий максимум на префиксе. Это поддерживается с помощью хранения количества вхождений каждого значения, а также используя факт, что максимум может только увеличиться, когда мы увеличиваем префикс. Аналогично обработаем зеркальный случай с перебором суффиксов первого блока. Итого два соседних блока мы обработали за $O(n)$, а значит итого мы нашли все хорошие подмассивы за $O(n)$. Теперь сохраним для каждой позиции все хорошие подмассивы и отсортируем их. Тогда, чтобы отвечать на запросы, требуется найти бинарным поиском нужную позицию и взять длину оставшегося суффикса хороших подмассивов в данной позиции как ответ.

Итоговая асимптотика $O(n + q \log n)$, что набирает 100 баллов. Также существует авторское решение, которое сортирует запросы сортировкой подсчетом и отвечает на них offline за $O(n + q)$ двумя указателями.

Разбор задачи «Инопланетные омофоны»

Автор задачи и разработчик: Александр Заварин

Первым делом определим пары одинаковых звуков. Представим, что звуки - вершины графа, пары одинаковых звуков - ребра. В таком случае нам нужно определить компоненты связности. Это можно определить простым проходом dfs, изначально присваиваем всем звукам индекс 0 (без индекса), идем по звукам, если у звука индекс 0, то запускаем dfs из этой вершинки, проставляя всем посещенным вершинам при этом одинаковый индекс, который до этого не выдавался другим группам одинаковых звуков. Таким образом получим массив индексов для каждого звука, по которым

легко определить, считается ли пара звуков одинаковыми. Будем называть полученные индексы - индексами звучания, чтобы не путать с номером (индексом) звука в самом наборе.

Для того чтобы набрать хоть какие то баллы, переберем каждый запрос, будем идти по подстрокам в запросе одновременно. Пусть мы прочитали какую-то часть у этих подстрок. Переберем каждый звук, среди них простым проходом с каждой позиции в подстроках проверим вхождение этого звука. При проверке будем поддерживать номер звука, максимального по длине. Проверим, одинаковые ли индексы звучания у наших звуков - если нет, значит эти подстроки не омофоны и ответ «`NO`», иначе сместим указатель в каждой на длину соответствующего звука и повторим. Если в какой то момент мы прочитали одну подстроку полностью, то если подстроки являются омофонами, то и во второй мы тоже должны были дойти до конца, проверяем это и выводим ответ на запрос. Это решение проходит 1 подгруппу, асимптотика: $O(q \cdot |t| \cdot S)$.

Рассмотрим, как решить подгруппы, для которых $b_i = d_i = |t|$, это значит, что все подстроки в запросах являются суффиксами текста.

Для 2 подгруппы поймем, что мы повторяем некоторые действия: для каждой позиции в тексте номер звука, который бы прочитали инопланетяне (начиная с этой позиции), всегда определен однозначно, в прошлом решении мы могли его считать несколько раз в разных запросах. Тогда для каждой позиции предподсчитаем номер звука, который входит с этой позиции как подстрока в текст и максимален по длине. Теперь для каждого запроса просто будем "прыгать"по уже насчитанным звукам, проверяя их на одинаковость по звучанию и перемещая указатель на длину звука для каждой подстроки. Такое решение имеет асимптотику: $O(|t| \cdot S + q \cdot |t|)$.

Заметьте, что такое решение не пройдет 1 подгруппу, так как мы пользуемся свойством $b_i = d_i = |t|$. В общем случае предподсчет максимального звука для каждой позиции (будем называть это **максимальным прыжком** для каждой позиции) не всегда соответствует макс. прыжку в подстроке в этой же позиции (так как подстрока может заканчиваться раньше и следовательно некоторые звуки просто в нее по длине не поместятся, хотя в тексте они могли бы поместиться).

Продолжим улучшать наше решение. Перейдем к 3 подгруппе.

Так как все подстроки в запросах являются суффиксами, то на самом деле в запросах всего максимум $|t|$ различных подстрок. Осталось придумать, как проверять очень быстро их на одинаковость по звучанию, очень хочется это делать аж за $O(1)$ после некоторого предподсчета. На помощь нам приходит хеширование. Давайте для каждого суффикса узнаем число звуков, которые мы выговариваем при его прочтении и полиномиальный хеш из индексов звучания для каждого звука, тогда потом для того чтобы два суффикса читались одинаково, достаточно проверить, что у них одинаковые число звуков и хеш индексов звучания. Эти две вещи и **будем называть хешем далее**. Теперь как это предподсчитать, в прошлой подгруппе мы научились предподсчитывать для каждой позиции в тексте максимальный "прыжок", тогда пойдем в тексте справа налево и последовательно для каждого суффикса насчитаем нужную нам информацию. Для суффикса длины 1 просто запишем, что у него 1 звук и хеш состоит из одного индекса звучания латинской буквы. Далее двигаемся влево, для каждой позиции мы знаем макс прыжок, а так как все суффиксы справа мы уже насчитали, то и оставшуюся часть суффикса мы тоже знаем, берем информацию у суффикса на расстоянии макс прыжка (макс звука) справа и обновляем хеш, добавляя туда индекс звучания нашего макс звука, а число звуков увеличиваем на 1. Таким образом мы получим нужную информацию про каждый суффикс и будем обрабатывать каждый запрос за $O(1)$. Итоговая асимптотика: $O(|t| \cdot S + q)$.

Для 4 подгруппы осталось придумать, как быстро насчитать макс прыжки для каждой позиции в тексте. Вспомним замечательный алгоритм Ахо-Корасик, развернем каждый звук и на полученных строках построим бор. Посчитаем на боре суффиксные ссылки как в Ахо-Корасике, а также терминальные суффиксные ссылки из каждой вершины в ближайшую терминальную (терминальная вершина бора - вершина, где заканчивается какой-то звук). Теперь пойдем по тексту справа налево и будем перемещаться по нашему бору, как в алгоритме поиска строк в тексте: пытаемся сделать переход по букве, если не выходит, прыгаем по суффиксной ссылке и повторяем попытку. Пусть мы прошли какую-то часть текста, параллельно перемещаясь по бору, давайте посмотрим, куда для вершинки бора ведет терминальная суффиксная ссылка. Во-первых, если мы находимся в какой-то вершине бора, это значит, что если мы будем спускаться по бору вниз, то в тексте мы

будем перемещаться с текущей позиции по буквам вправо. Т.е. если мы находимся в некоторой позиции в тексте, то строка, полученная из букв при спуске по бору из текущей вершины до корня, будет входить как подстрока, начиная с текущей позиции в тексте. При переходе по суффиксной ссылке мы просто удаляем какое-то число символов в конце этой строки, поэтому при переходе по терминальной суффиксной ссылке мы попадем в вершинку бора такую, что строка из букв, начиная с этой вершинки и до корня, входит как подстрока, начиная с нашей позиции в тексте, она является звуком и при этом длина максимальна, это ровно то, что нам нужно. Предподсчет суффиксных ссылок работает за $O(26 \cdot S)$, проход по тексту суммарно за $O(|t|)$, оставшаяся часть решения такая же, как в прошлой подзадаче, итого асимптотика: $O(26 \cdot S + |t| + q)$. Заметьте, что в каждой позиции мы получаем звук максимальной длины из одной терминальной суффиксной ссылки, не прыгая по суффиксным ссылкам дальше, как в обычном алгоритме Ахо-Корасика для поиска всех вхождений, поэтому число вхождений подстрок никак не влияет на асимптотику.

Теперь рассмотрим 5 подгруппу. В ней во всех запросах все подстроки являются префиксами текста. Давайте вернемся к более простому решению из 3 подгруппы, за квадрат для каждой позиции в тексте найдем макс. прыжок. Теперь рассмотрим, как выглядит последовательность звуков для некоторого префикса: мы прыгаем по макс прыжкам, набирая параллельно хеш, в какой то момент макс. прыжок будет перепрыгивать границу нашего префикса, получается на самом деле все, чего нам не хватает - это того, как выговаривается какой то префикс нашего звука, который слишком длинный, если бы мы знали эту информацию, то мы бы легко смерджили два хеша (вспомните, что мы дополнительно храним число звуков, поэтому для мерджа нам надо просто один из хешей умножить на основание хеша в степени числа звуков). В таком случае мы хотим посчитать хеш для каждого префикса у наших звуков (разве что для префикса, являющегося самим звуком, считать это не надо, для них хеш - это просто индекс звучания нашего звука). Воспользуемся динамическим программированием: отсортируем все строки по длине, для звуков - букв латинского алфавита хеш префикса очевиден, далее предположим, мы предподсчитали дп для всех строк длины $< k$. Теперь пусть мы рассматриваем некоторый звук длины k, пусть мы находимся в начале этого звука, тогда за $O(S)$ мы можем определить макс. звук, который мы можем выговорить при этом, не выговаривая весь звук полностью. Важно заметить, что мы никогда не должны залезать на последнюю позицию звука, так как звук выговаривать полностью мы и так умеем. Заметим, что все звуки, которые мы будем выговаривать, будут длины $< k$, следовательно, для них насчитано дп, а значит, мы можем просто перенести результаты префиксов звука, который мы выговариваем с первой позиции, в наше дп. Заведем глобальный хеш и запишем туда наш выговоренный звук, сместимся в нашем изначальном звуке длины k на длину выговоренного и повторим эти же действия, только при переносе результатов дп будем их мерджить с глобальным хешем. Таким образом за $O(S^2)$ предподсчитаем наше дп для всех звуков. Теперь, чтобы посчитать хеш для каждого префикса текста, также заведем глобальный хеш (изначально равный 0) начнем с первой позиции, за $O(S)$ найдем макс. звук, перенесем результаты его дп для каждого префикса, а его добавим в глобальный хеш, сместимся и повторим те же действия, только при переносе результатов дп будем их мерджить с глобальным хешем. Далее отвечаем на каждый запрос за $O(1)$, получаем асимптотику: $O(|t| \cdot S + S^2 + q)$.

Теперь вернемся к общему случаю задачи.

Для решения 6 подгруппы мы хотим для каждой подстроки предподсчитать ее хеш, чтобы так же, как и в прошлых подгруппах, обрабатывать запросы за $O(1)$. Для каждой позиции в тексте заведем битовый массив размера |t|, в k-й ячейке будет лежать 1, если начиная с этой позиции есть звук длины k, который входит как подстрока, это легко посчитать за $O(|t| \cdot S)$. Теперь переберем левую границу подстроки l, далее будем перебирать правую границу от l до $|t|$, поддерживая при этом хеш подстроки. Для подстроки длины 1 хеш очевиден, так же мы будем отмечать позиции важными, если на данный момент начиная с них мы выговариваем звук при прочтении подстроки. Пусть мы обработали уже какое то число правых границ и у нас есть хеш подстроки $[l, r-1]$, а так же отмечены важные позиции, хотим перейти к подстроке $[l, r]$. Переберем все важные позиции от l до r, среди них выберем такую наименьшую позицию i, что из i позиции можно сказать звук длины $r - i + 1$, если такая есть, то обновим хеш, убрав из нее хеш всех звуков, которые мы выговариваем, начиная с i позиции (и убрать маркеры важных позиций для них, разве что кроме позиции i) и добавим новый звук, который говорим из i позиции, если такой позиции нет, значит мы

добавляем в наш хеш звук-букву из позиции r и отмечаем ее важной. Тогда получаем решение за куб: $O(|t|^3 + |t| \cdot S + q)$. Так же возможны другие вариации решения, доказательство этого остается читателю в качестве упражнения.

Для 7 подгруппы используем идею из 5 подгруппы для предподсчета дп у всех звуков за $O(S^2)$ и так же для каждой позиции в тексте предподсчитаем макс. прыжок, но в этой подгруппе будем обрабатывать запросы онлайн. Для запроса нам для каждой подстроки надо узнать хеш. Для того чтобы для подстроки узнать хеш изначально поставим его равным 0, начнем с первой позиции, если макс. прыжок не перепрыгивает правую границу подстроки, добавим его в хеш и сдвинемся на длину прыжка, если полностью набрали подстроку, остановимся, мы получили нужный хеш, иначе как только найдем прыжок, который перепрыгивает границу, узнаем результат дп для нужного префикса в нем и смерджим хеш из дп с нашим. Таким образом получаем решение за $O(|t| \cdot S + S^2 + q \cdot |t|)$.

Для 8 подгруппы воспользуемся идеей с Ахо-Корасиком из 4 подгруппы, чтобы за $O(26 \cdot S + |t|)$ предподсчитать все макс. прыжки в тексте. Так же нам надо научиться считать дп для звуков быстрее квадрата. Для этого нам нужно оптимизировать поиск макс. звука при подсчете дп в самом звуке. Заметим, что похожее мы уже делали в тексте, бор для Ахо-Корасика у нас уже есть, тогда давайте просто для каждого звука пройдемся с права налево, начиная с предпоследней позиции (это важно, так как нам нельзя залезать на последнюю букву) и для каждой позиции предподсчитаем макс. звук как для обычного текста, на это мы потратим $O(S)$ действий. Для запросов искать хеш подстрок будем так же, как и в 7 подгруппе, из-за этого в нашей асимптотике появится $O(q \cdot |t|)$, итого получаем: $O(26 \cdot S + |t| + S + q \cdot |t|) = O(26 \cdot S + q \cdot |t|)$.

Чтобы сдать 9 и 10 подгруппы, осталось придумать, как быстрее обрабатывать наши запросы. Для каждой позиции в тексте мы знаем макс. прыжок, тогда давайте построим разреженную таблицу на этих прыжках, которая будет для каждой степени двойки 2^k хранить хеш из 2^k звуков, которые мы выговариваем при этих 2^k прыжках и позицию, в которой мы после этих прыжков окажемся, на предподсчет мы потратим $O(|t| \log |t|)$. Тогда теперь в запросе мы можем для каждой подстроки за $O(\log |t|)$ найти позицию, после которой макс прыжок перепрыгивает границу и так же хеш макс. звуков, которые мы выговаривали по пути к этой позиции. Получим потрясающую асимптотику: $O(26 \cdot S + |t| \cdot \log |t| + q \cdot \log |t|)$.

В 4, 8 и 9 подгруппы можно сдать оверкилл решения, в которых в асимптотике добавляется $O(S \cdot \sqrt{S})$ или $O(|t| \cdot \sqrt{S})$. Решение в Ахо-Корасике не разворачивает звуки и пользуется фактом, что когда мы находимся в некоторой вершине бора, переходов до корня по терминальным суффиксным ссылкам будет не больше $\sqrt{S}$.

Так же есть красивое решение, которое проходит все подгруппы с ограничением $|t| \leqslant 6000$. В нем мы сохраняем все звуки в бор, далее для каждой позиции за $O(|t|^2)$ узнаем все звуки, которые можно выговорить с этой позиции и сортируем их по длине. Мы хотим для каждой подстроки узнать ее хеш. Будем перебирать начальную позицию i, теперь запустим рекурсивную функцию $func$ следующего вида: она принимает хеш h, l - позицию, с которой мы хотим выговорить звук, r - правую границу, дальше которой мы не можем выговаривать (изначально положим хеш $h = 0$, $l = i$, $r = |t|$). Теперь что мы будем делать в функции, возьмем минимальный звук, который мы можем выговорить с позиции l (это звук из одной буквы), добавим в хеш h этот звук и запишем для соответствующей подстроки $[i, l]$. Теперь если в этой же позиции l есть следующий звук большего размера k, тогда запустим нашу же функцию от следующих аргументов: $func(h + (\text{звук из одной буквы}), l + 1, l + k - 2)$, она посчитает ответы для всех подстрок от $[i, l + 1]$ до $[i, l + k - 2]$, для подстроки $[i, l + k - 1]$ хеш - это хеш h, в который добавили наш звук длины k. Теперь посмотрим, есть ли с этой же позиции l еще больший звук, пусть он есть и он длины p, тогда повторим тоже самое запустим функцию $func(h + (\text{звук длины k}), l + k, l + p - 2)$ и далее опять посмотрим, есть ли звук большего размера. Если в какой то момент звука большего размера не оказалось или он залезает дальше границы r, просто запустим функцию с ограничением r: $func(h + (\text{звук максимальной длины f, не залезающий за границу}), l + f, r)$. Это работает за $O(|t|^2)$. Итоговая асимптотика $O(|t|^2 + S + q)$.

Разбор задачи «Сверхбыстрый поезд»

Автор задачи и разработчик: Даниил Васильев

Построим граф, где вершины являются городами, а рёбра являются железными дорогами. Заметим, что ответ всегда 0 или 1, так как мы можем построить простой путь от 1 до n, где каждое нечётное ребро берём со знаком '+', а каждое чётное со знаком '-'.

Далее, граф может быть одного из двух видов:

1. Граф не содержит циклов нечётной длины. Тогда граф двудольный. Пусть вершина 1 находится в первой доле (если нет, меняем доли местами). Заметим, что из-за этого текущее время всегда чётно в первой доле и нечётно во второй доле. Тогда здесь также будет работать простой путь, так как чётность его рёбер равна чётности ответа.

2. Граф содержит цикл нечётной длины. Тогда мы проверяем, какой ответ выдаст простой путь. Если он выдаст ответ 1, то мы можем его улучшить, дойдя до цикла нечётной длины, пройдя его полностью с суммарным временем -1, и вернувшись назад на простой путь. В результате время уменьшается до 0, что является наиболее оптимальным вариантом.

Итого, определяем, есть ли в графе цикл нечётной длины, и обрабатываем соответствующий случай, что занимает $O(n+m)$.

Разбор задачи «Классическая задача на дерево»

Автор задачи и разработчик: Андрей Павлов

Формально в задаче требовалось считать количество вершин w на пути (u, v) в дереве таких, что $dist(u, w) = a_w$, а также обрабатывать запросы обновления значения в точке.

Рассмотрим подгруппу 5, в ней $c_i \leqslant 2$, значит деревья в этой подгруппе являются бамбуками. Тогда пронумеруем каким-то образом вершины бамбука по расстоянию до любого из концов. Пусть b_u — это номер вершины u. Мысленно расположим все вершины в порядке возрастания номеров, понятно что мы получим массив где соседние элементы соединены ребром в дереве. Тогда расстояние между вершинами u, v ничто иное как $|b_u - b_v|$. От нас требовалось считать количество таких w на пути u, v, что $dist(u, w) = |b_u - b_w| = a_w$. Так как все w находятся слева или справа от u, то для всех w модуль раскрывается одинаковым образом. Не теряя общности будем считать что v находится правее u, то есть $b_v > b_u$. Тогда осталось посчитать количество таких w, что $b_u = b_w - a_w$. Давайте с данной идеей перейдем к более общему случаю.

Пусть во всех запросах $v_i = 0$, что соответствует 6 подгруппе. Давайте сделаем вершину 0 корнем дерева, тогда для ответа на запрос нужно прыгать из вершины u_i в корень дерева. Существует простой способ подсчета расстояния между двумя вершинами, одна из которых предок другой. Заметим, что когда мы прыгаем вверх то делаем это по предкам изначальной вершины u_i. Для подсчета расстояний насчитаем с помощью поиска в глубину расстояния до корня дерева из каждой вершины. Обозначим за h_i расстояние от вершины i до корня. Тогда расстояние между вершинами u, v где u — предок v, это $h_v - h_u$. Тогда в запросе требуется найти все такие вершины w среди предков, что $h_u - h_w = a_w$, то есть $h_u = h_w + a_w$.

Пусть запросы изменения отсутствуют, тогда давайте запустим поиск в глубину из корня. В момент когда мы заходим в вершину u в стеке рекурсии хранятся все ее предки, давайте хранить глобально какую-либо структуру в которой будем хранить значения $h_w + a_w$ для предков и количество таких значений, например подойдет структура map. Теперь когда мы в поиске в глубину хотим перейти к детям вершины, добавим ее в структуру, после выхода из этой вершины удалим ее из структуры. Тогда чтобы ответить на запрос для вершины u достаточно найти количество значений h_u в структуре.

Теперь решим задачу, когда существуют запросы изменения в точке. Давайте заметим, что обращений чтения к нашей структуре не более чем $O(q)$, а также и запросов изменения не более чем $O(q)$. Тогда давайте хранить в структуре в ключе x для каждого момента времени от 1 до q количество вершин, которые лежат в ключе x в этот момент времени. Но понятно, что тогда мы сохраним $O(q^2)$ информации, но заметим что нам интересны только те моменты времени, когда внутри структуры ключа x происходило какое-либо изменение, а таких не более чем запросов изменения, то есть $O(q)$

суммарно. Тогда давайте считать запросы и запомним для каждой вершины до всех изменений и после каждого изменения в каком месте и в какой момент времени она будет лежать в структуре, то есть рассмотрим значение $a_w + h_w$ для вершины до изменений и после каждого изменения. После чего возьмем каждое значение, которое встречается в структуре и сожмем все значения изменений внутри него по моментам времени данных изменений. Теперь внутри ключа x давайте хранить другую подструктуру, которая умеет прибавлять 1 и -1 на отрезке и узнавать значение в конкретной точке. Для этого подойдет дерево отрезков или дерево Фенвика.

Как теперь отвечать на запросы? Давайте для запроса сохраним момент времени, когда он был спрошен. Тогда для запроса i нам все еще нужно обратиться в структуру по ключу h_{u_i} и узнать количество вершин со значением $h_w + a_w$ в момент времени i, для этого в сжатой структуре найдем последнее возможное изменение и посчитаем в нем значение в точке. Когда нам нужно добавить какую-то вершину в структуру, то нужно рассмотреть когда происходило последнее изменение в данной вершине, пусть он происходил в момент j (если это первое изменение то $j = 0$), тогда в моменты времени $[j, i)$ данная вершина не менялась, тогда в структуре по ее значению в эти моменты времени прибавим 1 на отрезке $[j, i-1]$. Когда мы выходим из вершины откатим все изменения. Таким образом мы научились решать полностью подгруппу 6. Оценим текущую асимптотику решения: для чтения после сжатия структуры мы сделаем $O(q \log q)$ действий, когда мы заходим в вершину в поиске в глубину, то за $O(\log q)$ обрабатываем каждое изменение и каждый вопрос, суммарно $O(q \log q)$. Тогда асимптотика $O(n + q \log q)$.

Чтобы решить задачу в общем случае нужно обобщить текущее решение на другие запросы. Обозначим $t = \text{lca}(u, v)$. Так как мы подвесили дерево, то любой простой путь из вершины u в вершину v будет идти сначала вверх в сторону корня, пока не дойдет до t, после чего пойдет вниз в сторону вершины v. Давайте рассмотрим путь от lca(u, v) до v, на нем текущая длина пути уже $h_u - h_t$, так как мы идем вниз от вершины t, то длина пути до вершины w между t и v равна $h_w - h_t$, а значит нужно искать такие вершины w, что $h_u - 2 \cdot h_t + h_w = a_w$, то есть $h_u - 2 \cdot h_t = a_w - h_w$. Это обрабатывается следующим образом: создадим точно такую же структуру, которую точно также сожмем, только будем хранить в ней уже значения вершин по $a_w - h_w$, а спрашивать нужно будет в точке $h_u - 2 \cdot h_t$. Теперь разобьем путь u, v в 4 других пути, которые прикреплены к корню. То есть чтобы обработать вершины на пути от u до t давайте посчитаем количество нужных вершин на пути $u \to 0$ и $t \to 0$, после чего вычтем из ответа на первый запрос вершины, которые не лежат на пути $u \to t$, то есть ответ на второй запрос (саму вершину t мы обработаем отдельно, потому что в ходе данных операций она не участвует в пути, что неправда). Теперь разобьем подобным образом оставшийся путь от t до v на два других $0 \to t$ и $0 \to v$ и также вычтем один из другого. Так как запросы из корня мы умеем обрабатывать за $O(\log q)$, то 4 запроса к корню мы тоже обработаем за $O(\log q)$.

Итоговая асимптотика: $O(n + q \log q)$, что набирает 95-100 баллов в зависимости от реализации.

Разбор задачи «Цветной диаметр»

Автор задачи: Никита Голиков, разработчик задачи: Игорь Маркелов

Для решения задачи требуется решить следующие 2 подзадачи:

1. Научиться как можно более эффективно искать оптимальное время.

2. Научиться как можно более эффективно вычислять функционал для фиксированного времени.

Для решения первой идейной части требовалось заметить, что функция от времени выпукла. Доказательство этого факта остается читателю в качестве интересного, но несложного упражнения. Соответственно к ней можно применить стандартные способы поиска минимума выпуклой функции, например, тренарный поиск в целых числах или тренарный поиск в числах с плавающей точкой с использованием оптимизации с помощью золотого сечения.

Для решения второй требовалось в начале научиться решать для двух цветов, а затем обобщить это решение на количество цветов по порядку равное количеству точек.

Задача для двух цветов формулируется следующим образом: даны 2 набора точек, требуется посчитать наибольшее расстояние между точками разных цветов. Нетрудно показать, что существует пара с наибольшим расстоянием, такая что точки лежат на выпуклых оболочках данных множеств, и, при этом, если расширить оба множества до их выпуклых оболочек, наибольшее расстояние не изменится. Чтобы посчитать расстояние между парой выпуклых многоугольников, можно воспользоваться суммой Минковского, а именно сложить первый многоугольник со вторым, отраженным относительно начала координат (некоторые называют этот прием разностью Минковского, так как по смыслу получается, что в итоговом множестве лежат все вектора разностей точек из двух множеств). В этой сумме нас интересует самый длинный вектор, то есть наиболее удаленная от начала координат вершина. Расстояние до нее и будет ответом. Таким образом можно для двух множеств с суммарным размером $O(n)$ найти ответ за $O(n \cdot \log n)$, в случае, если точки не отсортированы, и за $O(n)$, если точки отсортированы лексикографически (воспользовавшись алгоритмом Эндрю для поиска выпуклой оболочки).

Задачу для нескольких цветов можно решить разными способами. Предлагается рассмотреть 2 из них.

1. Переберем бит в двоичной записи номеров цветов, в котором различаются цвета, точки которых являются оптимальными. Отнесем к первому и второму множеству все точки, в цвете которых зафиксированный бит равен 0 и 1 соответственно. Таким образом мы рассмотрим все возможные пары точек различных цветов и только их (пусть и некоторые по несколько раз).

2. Решим задачу с помощью метода "Разделяй и Властвуй"на множестве цветов. Будем сливать 2 множества точек в 1, перед этим посчитав для них оптимальный ответ. Таким образом, так же будут рассмотрены все пары точек разных цветов. Чтобы стратегия сливания случайно была не квадратичной, можно сливать 2 множества с самым маленьким размером, или же построить бинарное дерево на цветах.

Частичные баллы в этой задаче набирались различными комбинациями идей, описанных выше. Итого, с помощью этих идей можно получить время работы $O(n \cdot \log T \cdot \log^2 n)$.

Чтобы получить $O(n \cdot \log T \cdot \log n)$ и 100 баллов, требовалось заметить, что точки можно отсортировать 1 раз за подсчет функционала.