Разбор задачи «Перфокарты»

Автор: Дмитрий Саютин
Разработчик: Рамазан Рахматуллин

Первая подзадача предназначена для переборного решения, пробующего все возможные порядки выкладывания карт.

Посмотрим на самую верхнюю перфокарту в ответе. Все присутствующие в ней символы должны совпадать с соответствующими символами строки s. Теперь рассмотрим вторую сверху перфокарту. Все позиции в ней, за исключением тех, которые присутствовали в первой перфокарте, должны также совпадать с символами строки s. Данную идею можно обобщить: будем брать любую перфокарту, которую в данный момент можно положить под уже выбранные, и помечать непомеченные позиции этой перфокарты в строке s. После этого действия некоторые перфокарты можно будет положить под уже выбранные, причем если перфокарту можно положить, то это свойство останется.

Из этой идеи вырисовывается решение. Для каждой перфокарты храним число непомеченных позиций, которые не совпадают со строкой s. Также храним любым образом множество перфокарт, у которых это число равно нулю. Повторяем следующую процедуру:

- достаём любую перфокарту из множества, если оно непусто
- перебираем ее непомеченные позиции (они по предположению все совпадают с соответствующими символами строки)
- каждую непомеченную позицию помечаем
- уменьшаем число на единицу у всех перфокарт, у которых на этой позиции символ не совпадает с правильным
- если у какой-то перфокарты число обратилось в 0, добавляем её в множество

Если в множестве не осталось перфоркат, при этом не все из них положены, то ответ — -1. Также нужно не забыть про ситуацию, когда все карты выложить получилось, но при этом некоторые позиции остались непомеченными; это означает, что какая-то позиция не содержит нужной буквы ни в одной перфокарте, поэтому ответ также -1.

Осталось доказать, что если наш алгоритм не нашёл ответ, то его не существует. Предположим, что наш алгоритм зашёл в тупик, выложив перфокарты $p_1, \ldots, p_k$, при этом существовал какой-то иной порядок $q_1, \ldots, q_n$, реализующий заданную строку. Но тогда можно рассмотреть первую перфокарту q_j во втором порядке, которая не встречается среди чисел p_i — легко понять, что последовательность $p_1, \ldots, p_k, q_j$ является корректной, т.к. q_j можно было положить даже после некоторого подмножества предшествовавших перфокарт.

Вторая подзадача допускает неэффективные реализации идеи выше (например, вместо поддержания множества «нулевых» перфокарт, пробегаться по всему списку оставшихся перфокарт и искать подходящую).

Эффективная версия решения работает за время $O(n+m+\sum_{i=1}^{n} k_i)$ и проходит на полный балл.

Разбор задачи «Родные просторы»

Автор и разработчик: Николай Будин

Дана строка s. За ход можно взять два соседних символа $s[i]$ и $s[i+1]$, если $A[s[i]][s[i+1]]$, и удалить из строки $s[i+1]$. Нужно получить лексикографически минимальную возможную строку.

Рассмотрим граф, вершины которого соответствуют символам исходной строки. Когда берем пару соседних символов и удаляем правый, проведем в этом графе ориентированное ребро из исходной позиции левого символа в исходную позицию правого. Заметим, что получился подвешенный лес. Корнями деревьев являются те и только те символы, которые остались в итоговой строке. Вершины одного дерева образуют отрезок подрядыдущих символов, корень является самым левым их них.

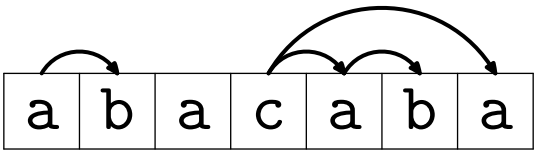

Рис. 1: Граф для первого теста.

Задача состоит из двух частей — в первой нужно разобраться с тем, какие строки вообще возможно полчить такой процедурой, а во второй — научиться искать минимальный лексикографически ответ.

Часть 1

Рассмотрим символ c, который останется в итоговой строке. Будем говорить, что отрезок символов, удаленных непосредственно после c, был удален с использованием c. Например, на картинке отрезок «caba» был удален с использованием «c». Заметим, что если отрезок $s[l+1 \dots r]$ можно удалить с использованием $s[l]$, то также с помощью $s[l]$ можно удалить и $s[l+1 \dots r-1]$. Поэтому, для каждой позиции $s[l]$ нужно найти максимальную длину отрезка $s[l+1 \dots r]$, который можно удалить с использованием $s[l]$.

Способ 1.1

Можно воспользоваться методом динамического программирования. $dp[c][l][r]$ — можно ли удалить отрезок $s[l \dots r]$ с использованием символа c, написанного ровно перед ним. Если $A[c][s[l]] = 0$, то это в любом случае невозможно. Иначе, переберем m ($l \leqslant m < r$). Если $dp[c][l][m]$ и $dp[c][m+1][r]$, то $dp[c][l][r] = true$. Второй вариант — если $dp[s[l]][l+1][r]$, то $dp[c][l][r] = true$. Время работы такого алгоритма $O(n^3 \cdot k)$.

Способ 1.2

Можно дополнительно ускорить предыдущий алгоритм. $dp'[c][l]$ — максимальное r, для которого $dp[c][l][r] = true$. Пересчет происходит аналогично. Для фиксированных c и l, инициализируем $dp'[c][l] = dp'[s[l]][l+1]$, и перебираем r в порядке возрастания. Если $r \leqslant dp'[c][l]$ и $A[c][s[r]]$, то $dp'[c][l] = \max(dp'[c][l], dp'[c][r])$. Получился алгоритм за $O(n^2 \cdot k)$.

Способ 2

Это решение будет отличаться от предыдущих. Заведем стек st и пройдемся по строке s справо налево. Когда рассматриваем $s[i]$, пока стек не пуст и $A[s[i]][s[st.top()]]$, удалим верхний элемент из стека. Оказывается, что после этого максимальный отрезок, который можно удалить, используя $s[i]$, заканчивается в $s[st.top() - 1]$ (либо в n, если стек пуст). После этого, добавим в стек i. Этот алгоритм работает за линейное время.

Докажем этот алгоритм. Пусть для какого-то i длина отрезка была найдена неправильно. Рассмотрим самый правый из таких i. Мы решили, что отрезок заканчивается в y ($y = st.top() - 1$) и $A[s[i]][s[y+1]] = 0$, а на самом деле отрезок должен заканчиваться в r ($y < r$). Рассмотрим дерево способа, в котором мы удалили отрезок до r. В этом способе $s[y+1]$ не может быть непосредственным ребенком $s[i]$. Будем подниматься из $s[y+1]$ вверх по дереву, пока не дойдем до ребенка $s[i]$, пусть это будет $s[x]$. По предположению, для $s[x]$ мы нашли отрезок корректно. А значит, он должен заканчиваться не раньше $s[y+1]$, а значит $y+1$ уже был удален из стека, когда мы рассматривали $s[x]$, значит его не могло быть в стеке и когда мы рассматривали $s[i]$. Противоречие.

Часть 2

Теперь осталось разбить строку s на отрезки a_1, a_2, $\dots a_k$, такие что весь отрезок a_i, кроме первого символа, может быть удален с использованием $a_i[1]$. А строка $a_1[1] + a_2[1] + \dots + a_k[1]$ — лексико-

графически минимальная возможная, где + — операция конкатенации строк.

Способ 1

Воспользуемся методом динамического программирования. $ans[i]$ — ответ для строки $s[i \dots n]$. Тогда пересчет $ans[i] = \min(ans[i], s[i] + ans[j])$, если $s[i+1 \dots j-1]$ можно удалить с использованием $s[i]$. Такой алгоритм работает за $O(n^3)$.

Способ 2

Вместо того, чтобы явно хранить в $ans[i]$ строки, сохраним $next[i]$ — индекс второго символа $ans[i]$. Тогда $ans[i] = s[i] + s[next[i]] + s[next[next[i]] + \dots$. Ссылки $next[i]$ образуют подвешенное дерево. А каждое $ans[i]$ — путь из вершины i до корня. В этом дереве можно вычислить двоичные подъемы, и для каждого двоичного подъема хеш. Чтобы сравнить лексикографически две строки, нужно найти их максимальный совпадающий префикс, и сравнить следующий символ. Найти максимальный совпадающий префикс двух путей в дереве можно за $O(\log(n))$ с помощью двоичных подъемов и хешей.

Способ 2.1

Если применить такое сравнение строк к первому способу, получится алгоритм за $O(n^2 \log(n))$.

Способ 2.2

Построим дерево отрезков на $ans[i]$, которые мы сравниваем с помощью двоичных подъемов и хешей. Тогда $ans[i] = s[i] + \min\limits_{j=i+1}^{dp'[s[i]][i+1]} ans[j]$, это один запрос к дереву отрезков. Получается $O(\log^2(n))$ на один запрос. И алгоритм работает за $O(n \log^2(n))$.

Способ 2.3

Объединим алгоритмы первой и второй части. В тот момент, когда мы рассматриваем $s[i]$ и выкидываем из стека индекс $st.top()$, сделаем $ans[i] = \min(ans[i], s[i] + ans[st.top()][1 \dots])$. Такой алгоритм сделаем $O(n)$ сравнений и будет работать за $O(n \log(n))$.

Заключение

В зависимости от способа, выбранного в первой и второй части, решение может получить разные баллы. Полное решение работает за $O(n \log(n))$.

Разбор задачи «Игра с тайным смыслом»

Автор: Михаил Дворкин
Разработчики: Павел Кунявский, Михаил Дворкин

В отличие от большинства задач олимпиады, в этой задаче нет четкого разделения, какое количество баллов можно получить за различные идеи решения. Кроме того, скорее всего будет много решений, сильно отличающихся от авторских, да и авторские решения достаточно сильно могут зависеть от деталей реализации и подбора использующихся в них констант. Из-за необычного формата, задача также отличалась сложностью отладки решения, поэтому важно было не бросаться сразу писать сложное решение, а постепенно усложнять решение, экспериментируя с различными идеями и оптимизациями. Жюри надеется, что этот эксперимент понравился участникам.

Решение, которое передаёт один бит за игру, уже набирает 10 баллов. Для этого рассмотрим ход $(1, 2)$ или любой другой ход, который сразу же может сделать программа и в случае, когда решение ходит первым, и когда бот делает первый ход, а программа — второй. Тогда если нужно передать

бит 1, то программа делает этот ход, а следующим ходом сделает проигрышный ход $(1, 1)$. Для передачи бита 0 программа сразу же делает проигрышный ход $(1, 1)$.

Можно модифицировать это решение, научив его выигрывать партию. Рассмотрим два хода $(1, 2)$ и $(2, 1)$. Заметим, что решение всегда может сделать эти два хода в начале. Тогда мы можем передавать один бит при помощи порядка этих ходов. При этом такое решение ещё и будет всегда выигрывать игру, поэтому оно набирает 20 баллов. Интересной особенностью такого решения является то, что оно может игнорировать ходы бота и вообще не хранить игровое поле, но развивать дальше эту идею вряд ли получится.

Начнем с обсуждения, как выигрывать игры. Игра «Щелк» известна специалистам по теории игр одним интересным свойством — в этой игре всегда выигрывает первый игрок, но уже для небольших полей не известна выигрышная стратегия. Посмотрим на ход (n, n), закрашивающий правый верхний угол. Этот ход либо выигрышный, либо у второго игрока есть ответный выигрышный ход. Но в этом случае первый игрок может вместо хода (n, n) сделать этот ход и выиграть. Но это доказательство не позволяет найти выигрышную стратегию.

В случае квадратного поля (как в нашей задаче) есть симметричная выигрышная стратегия. Первый игрок может сделать ход $(2, 2)$, оставив от игрового поля только первый столбец и первую строку. После этого он будет симметрично повторять ходы второго игрока. В этой задаче решение может играть как первым, так и вторым, но так так бот делает случайные маленькие ходы, то бот не сможет сделать ход $(2, 2)$ в начале игры, поэтому решение может воспользоваться симметричной стратегией и даже если ходит вторым. При этом решение не обязано сразу же пользоваться симметричной стратегией, поскольку противник не делает большие ходы. Можно в начале игры передавать информацию, игнорируя ходы противника, а в конце игры переключится на симметричную стратегию, чтобы выиграть партию. Пока оставшиеся два ряда длинные, можно передавать по одному биту за ход выбором стороны и несколько бит, выбирая длину хода. Это позволяет передать несколько десятков бит за игру и получить 15–20 баллов и ещё 10 баллов за выигрыш всех игр.

Другой идеей является кодирование бит координатами хода. Например, можно каждым ходом кодировать 10 бит — 5 бит номером строки и 5 бит номером столбца. У этого подхода есть два недостатка. Во-первых, игра будет заканчиваться достаточно быстро, так как часто придётся делать ходы, закрашивающие много клеток. Во-вторых, если нужный ход сделать невозможно, нужно както указать, что этот ход не передаёт информацию на самом деле. Избавиться от этих недостатков можно, если передавать информацию только одной координатой хода, выбирая вторую координату так, чтобы закрашивать как можно меньше клеток. Это даёт возможность сделать больше ходов, передающих информацию, а если ход сделать нельзя, то решение делает ход $(2, 2)$ и переключается в режим симметричной стратегии для выигрыша игры. Таким способом можно получить 40–60 баллов в зависимости от деталей реализации.

Проблема описанных подходов в том, что в одном из них съедается много клеток, а в другом — слишком мало вариантов хода. Нужен промежуточный вариант! Заметим, что необязательно передавать информацию именно координатами. Можно перенумеровать все возможные ходы по какому-то правилу и таким образом передавать информацию. Выберем константу k и рассмотрим все ходы, которые закрашивают не более k клеток. Пусть количество таких возможных ходов оказалось равно C. Тогда если оба запуска перенумеруют все эти ходы одинаковым способом, то номером выбранного хода можно передать примерно $\log_2 C$ информации. В таком решении необходимо подобрать константу k. Если её выбрать небольшой, то будет передаваться мало информации за ход, а при большом k ходы быстро закончатся. Значение k можно подобрать экспериментально, запустив решение несколько раз с разными значениями k или отправив его в тестирующую систему. Например, в экспериментах жюри оптимальным был выбор $k \approx 10$, но это значение может быть разным в зависимости от деталей реализации. Подобные решения позволяют передавать около 700 бит за игру и получить 80–90 баллов, если добавить в это решение переход к выигрышной стратегии не слишком поздно. Одной из особенностью такого решения является то, что количество передаваемых бит за ход может быть различным в зависимости от значения C, которое зависит от игровой ситуации.

Улучшить это решение можно следующим образом. Заметим, что не все выбранные ходы одинаково ценны: большие ходы (закрашивающие много клеток) быстрее приводят к окончанию игры,

поэтому делать их невыгодно. Давайте передавать короткими ходами мало бит информации, а большими ходами — много бит информации, то есть "невыгодные" ходы будут полезны хотя бы тем, что они передают больше бит информации. Рассмотрим все рассматриваемые решением потенциальные ходы на данном шаге, пусть опять-таки их количество равно C. Ранее мы предлагали каждому из таких ходов сопоставить двоичную строку длины $\approx \log_2 C$. Теперь же сопоставим разным ходам разные двоичные строки: ходам, закрашивающим мало клеток будут сопоставлены короткие строки, а ходам, закрашивающим много клеток — длинные строки. Предположим, что решение в среднем передаёт s бит, закрашивая одну клетку. Константа s — эвристическая и в решениях жюри использовалось значение $s \approx 1{,}1$. Нам нужно для каждой из возможных C клеток выбрать двоичное кодовое слово w_i (передаваемая этим ходом информация), так чтобы максимизировать среднее значение величины $\text{len}(w_i) - e_i \times C$, где e_i — количество клеток, закрашенных этим ходом.

Чтобы сопоставить возможным ходам такие строки, можно воспользоваться эвристическим жадным решением, похожим на алгоритм построения кода Хаффмана (оптимального префиксного кода) или динамическим программированием. Например, в алгоритме Хаффмана строится представление множества строк в виде дерева, где у каждой вершины, не являющейся листом, — два потомка, а рёбрам соответствуют передаваемые биты 0 или 1. В листьях записаны ходы (выбранные клетки), а передаваемое кодовое слово w_i есть последовательность нулей и единиц на пути из корня дерева. Для построения дерева выбираются две вершины, для которых количество закрашиваемых клеток максимально, и они объединяются вместе. Это продолжается, пока не останется одна вершина — корень дерева.

Конкретные значения константы s, а также значения k (отсечения, насколько большие ходы допускаются), нужно подобрать экспериментально.

Разбор задачи «Доклад инвесторам»

Автор:	Ильдар Гайнуллин
Разработчик:	Егор Чунаев

Назовем дерево с числами на листьях *деревом поиска*, если возможно упорядочить детей у каждой нелистовой вершины, чтобы значения на листьях возрастали в порядке обхода.

Нам дано дерево из n вершин, где у каждой вершины не более чем k сыновей; требуется для каждого листа выбрать одно значение из множества возможных значений для данного листа, чтобы получилось дерево поиска. Обозначим за M суммарное количество возможных значений в листьях.

В первой подзадаче дерево достаточно маленькое, и максимальное количество сыновей не превосходит двух. Поэтому в ней работает полный перебор — мы можем перебрать все способы расставить все значения в листья и перебрать порядок детей в каждой из вершин. У нас есть 2^n вариантов упорядочить детей и примерно $2^{\frac{M}{2}}$ вариантов зафиксировать числа в листьях в худшей конструкции ($\frac{n}{2}$ листьев, в каждом два варианта), получаем $O(2^{n+\frac{M}{2}})$.

Во второй подзадаче n и M все ещё достаточно маленькие, но максимальное количество детей уже не ограничено. Заметим, что если числа в листьях зафиксированы, то порядок детей в каждой из вершин восстанавливается однозначно. Каждому поддереву можно сопоставить два числа — минимальное и максимальное значение в этом поддереве. Тогда в каждой вершине надо упорядочить детей по минимальному значению в поддереве и проверить что отрезки значений поддеревьев не пересекаются.

Для решения последующих подзадач нам надо воспользоваться методом динамического программирования.

Обозначим за $dp(v, x)$ минимально возможное максимальное значение в поддереве, если мы можем выбирать в этом поддереве только числа больше или равные x, либо $+\infty$, если это поддерево невозможно упорядочить в таких условиях.

Переменная x может принимать значения от 0 до максимального значения элемента по всем листьям M.

Обсудим, как инициализировать значение динамического программирования в листе. Если лист содержит числа $a_1 < a_2 < \ldots < a_r$, то

$$dp(v,0) = dp(v,1) = \ldots = dp(v,a_1) = a_1$$
$$dp(v,a_1+1) = dp(v,a_1+2) = \ldots = dp(v,a_2) = a_2$$
$$\ldots$$
$$dp(v,a_{r-1}+1) = dp(v,a_{r-1}+2) = \ldots = dp(v,a_r) = a_r$$
$$dp(v,a_r+1) = dp(v,a_r+2) = \ldots = dp(v,M) = +\infty$$

Обозначим детей вершины за $u_1, u_2, \ldots, u_m$. Тогда верно следующее:

$$dp(v,x) = \min_{\text{перестановка } p} dp\left(u_{p_1}, dp\left(u_{p_2}, dp\left(\cdots dp\left(u_{p_m}, x\right)\cdots\right)\right)\right)$$

Если сжать значения в листах, то эта динамика имеет $O(nM)$ состояний, поэтому это решение работает за $O(nM \cdot k!)$ и проходит все подгруппы с $k \leqslant 5$ и $n, M \leqslant 2000$, а также какие-то из групп с $k = 8$ и меньшими n.

Пересчет в вершине можно улучшить, введя вспомогательную динамику от подмножества уже взятых детей $S = \{a_1, a_2, \ldots, a_t\}$ и числа x, обозначающего текущее значение максимума. Положим:

$$f(S,x) = \min_{\text{перестановка } p} dp\left(a_{p_1}, dp\left(a_{p_2}, dp\left(\cdots dp\left(a_{p_t}, x\right)\cdots\right)\right)\right)$$

.

Такую динамику в вершине можно посчитать за $m2^m$, где m — количество детей вершины.

Вершин со степенью m не более чем $\frac{n}{m}$, поэтому это решение будет работать за $O(\sum_{m=1}^{k} M \cdot \frac{n}{m} m 2^m) = O(2^k nM)$

Такого метода пересчета уже достаточно, чтобы пройти все группы, где $n, M \leqslant 2000$.

Для решения трёх последних подзадач надо делать что-то аналогичное описанной динамике, но хранить значение минимума не по всем возможным значениям x, а использовать какой-то сжатый вид.

Для каждой вершины v будем хранить P_v: множество упорядоченных пар (a,b) таких, что возможно так выбрать значения в листьях в поддереве вершины v, чтобы получилось дерево поиска, и при этом минимальное и максимальное значение листьев равно a и b соотвественно.

Утверждение 1: Если есть две пары $(a,b) \in P_v$ и $(c,d) \in P_v$, что $a \leqslant c \leqslant d \leqslant b$ и $(a,b) \neq (c,d)$, тогда пару (a,b) можно выбросить из P_v.

Доказательство: Решение, в котором поддерево соответствует паре (a,b), можно поменять, заменив это поддерево на решение для пары (c,d). Тогда легко видеть, что дерево останется деревом поиска.

Утверждение 2: Для вершины v с детьми $c_1, c_2, \ldots, c_k$, можно найти P_v за $O((|P_{c_1}| + |P_{c_2}| + \ldots + |P_{c_k}|) \cdot (2^k + \log n))$

Доказательство:

Нам нужно сделать следующее, выбрать пары $(a_i, b_i) \in P_{c_i}$ так, чтобы эти пары не пересекались как отрезки, а затем добавить $(\min a_i, \max b_j)$ в P_v

Выпишем в один список все a_i и b_i, отсортируем и оставим среди них только различные; получим последовательность $s_1, s_2, \ldots, s_m$. После этого рассмотрим следующую динамику:

Значение $dp_{i,S}$: пусть мы уже выбрали непересекающиеся отрезки для детей из множества S, все они имеют $a_j \geqslant s_i$, и минимальное возможное значение $\max b_i$ среди них равно $dp_{i,S}$.

Затем, $dp_{i,S} = \min$ среди

- $dp_{i+1,S}$
- $dp_{j,S\cup\{t\}}$, если есть пара (s_i, s_j) в P_{c_t}
- j, если есть пара (s_i, s_j) в P_{c_t} и $S = \{t\}$

Мы можем найти значение этой динамики в порядке убывания i.

Затем, достаточно поставить $P_v = \{(s_i, s_{dp_{i,C}}) | 1 \leqslant i \leqslant m\}$, где C это множество всех детей.

Сортировка и динамика вместе работают за $O((|P_{c_1}| + |P_{c_2}| + \ldots + |P_{c_k}|) \cdot (2^k + \log n))$

После этого, как это следует из утверждения 1, можно удалить бесполезные пары из P_v. Это делается простым линейным проходом.

Также это можно сделать менее оптимально, просто перебирая все возможные порядки детей, тогда эта часть будет работать за $O((|P_{c_1}| + |P_{c_2}| + \ldots + |P_{c_k}|) \cdot (k! + \log n))$

Утверждение 3:

Обозначим за z_v суммарный размер множеств всех листьев в поддереве вершины v.

Пусть v это вершина с хотя бы двумя детьми.

Рассмотрим последовательность $a_1, a_2, \ldots, a_k$, где $a_i = z_{c_i}$

Тогда $|P_v| \leqslant 2 \cdot (\sum a_i - \max a_i)$

Доказательство:

Пусть j это сын с наибольшим значением a_j

Заметим, что каждая пара $(a, b) \in P_v$ имеет либо $a \notin P_j$ либо $b \notin P_j$.

А также, после удаления бесполезных пар, для каждого x в множестве содержится не более одной пары (x, b) и не более одной пары (a, x). Это значит, что каждый $i \neq j$ поспобствует добавлению не более двух пар в P_v. А из замечания выше мы знаем, что это покроет все пары в P_v.

Таким образом, $|P_v| \leqslant 2 \cdot (\sum a_i - \max a_i)$

Утверждение 4:

$|P_1| + |P_2| + \ldots + |P_n|$ это $O(n \log n)$, где n это сумма размеров всех списков листьев и количества вершин.

Доказательство:

Доказательство тривиально из стандартной техники «`меньшее к большему`» и утверждения 3.

В итоге можем найти P_v для всех вершин за $O(n \log n \cdot (2^k + \log n))$. После этого ответ «`Yes`» только если $|P_{root}| \neq 0$, а дальше требуется восстановить ответ, что является несложным техническим упражнением с помощью посчитанных динамик.

Менее оптимальные решения, использующие перебор все перестановок детей и работающие за $O(n \log n \cdot (k! + \log n))$, проходят все группы кроме последней.

Разбор задачи «Московские числа»

Автор: Николай Будин
Разработчики: Филипп Рухович, Дмитрий Саютин, Николай Будин

Если в заданной строке не было знаков вопроса, то значение числа найти легко — нужно лишь для каждой буквы в строке понять, есть ли справа от нее строго б**о**льшая буква, и прибавить или вычесть её номинал из ответа в зависимости от этого. Нахождение большей буквы перебором дает 6 баллов; однако если идти справа налево и хранить максимум на суффиксе, то можно набрать 15 баллов.

Третью группу тестов легко пройти, перебирая все возможные значения букв, которые можно найти под вопросиками, и выбирая оптимальный вариант. Отметим, что чтобы уложиться в ограничения по времени, необходимо вычислять значение каждого из получаемых чисел за линию.

Набрать же баллы по оставшимся двум группам помогает динамическое программирование. Действительно, пусть $dp[i][j]$, где $1 \leqslant i \leqslant n$, $0 \leqslant j < \Sigma$ — это максимально возможная стоимость i-го суффикса $S[i..n]$ строки S в предположении, что максимальной из букв на этом суффиксе является буква j. Будем вычислять такую динамику справа налево. При добавлении к строке слева i-го символа для определения знака его стоимости достаточно знать максимум среди символов от $i+1$ и дальше; в силу этого, любое $dp[i][j]$ может быть пересчитано за $O(\Sigma)$, где Σ — размер алфавита (в нашем случае 26). Таким образом, сложность такого решения $O(n\Sigma^2)$. Аккуратная реализация именно такого решения давала полный балл; однако отметим, что с помощью максимумов на префиксах и суффиксах массивов $dp[i][\ldots]$ длины 26 для разных i, решение можно было оптимизировать до $O(n\Sigma)$.

Разбор задачи «Гаджеты на дереве»

Автор: Максим Ахмедов
Разработчики: Арсений Кириллов, Азат Исмагилов

$m = 2n - 4$

Сначала обсудим решение в предположении, что $m = 2n - 4$. Заметим, что рёбра можно покрасить в чёрный и белый цвет таким образом, что в любом гаджете будет ровно одно чёрное и ровно одно белое ребро. Действительно, сначала покрасим в чёрные и белые цвета вершины леса так, чтобы у любого ребра концы имели разные цвета. После этого можно назначить ребру (u, v) тот же цвет, что у вершины u.

Заметим, что если бы в дереве были представлены все возможные $2n-2$ рёбер, ответ бы всегда существовал. Действительно, в таком ориентированном графе существует эйлеров цикл, который легко строится обходом в глубину. Этот цикл имеет чётную длину, значит он легко разбивается на $n-1$ гаджет.

Из этого наблюдения находится решения и для случая $m = 2n-4$. Построим тот же эйлеров цикл для полного дерева. Два удалённых ребра могут разбивать этот цикл либо на два отрезка чётной длины, либо на два отрезка нечётной длины. В первом случае также легко получить разбиение на гаджеты; в оставшемся случае легко видеть, что два удалённых ребра были одного цвета, а значит рёбер какого-то цвета среди оставшихся больше, чем другого; это доказывает, что ответ в таком случае – «невозможно».

Общий случай

Рассмотрим разбиение рёбер на пары (гаджеты) в ответе. Посмотрим произвольную вершину v и её поддерево; если наша вершина не является корнем, то рёбра между родителем p вершины v и самой v для удобства также включим в поддерево. Некоторые пары рёбер будут находиться целиком в поддереве, некоторые пары будут находиться целиком вне, а какие-то пары могут пересекать эту границу — одним концом в поддереве, другим снаружи. Заметим, что такие пары бывают только следующего вида: либо пара рёбер идущая вглубь нас $(u \to p, p \to v)$, либо пара рёбер идущая

наружу от нас ($v \to p, p \to u$), u обозначает какую-то вершину смежную с p, отличную от v. Обратите внимание, что u может быть как предком p, так и одной из детей p; кстати, в английском языке в такой ситуации используется удобный термин "sibling" (брат или сестра), вершина u является "sibling"-ом вершины v.

Однако заметим, что если у нас будет и пара внутрь, и пара наружу, то то можно было бы вместо этого организовать пару рёбер из двух нижних и отдельно пару из двух верхних. Поэтому можно запретить проводить две такие пары одновременно. Таким образом, получается что границу будет пересекать или ноль пар, или одна пара внутрь, или одна пара наружу. Причём заметим, что случай «ноль пар» от «одна пара в какую-то сторону» определяется просто чётностью количества рёбер в поддереве. На самом деле в процессе решения окажется, что ориентация пары (если она должна быть) тоже определяется однозначно.

Воспользуемся методом динамического программирования `dp[v] = 1/0` — можно ли разбить поддерево на пары. Если наша чётность подразумевает, что есть пара внутрь или наружу, разрешим себе «забрать» гипотетическое ребро вне поддерева в эту пару. На самом деле с учётом того, что `dp[v] = 0` для любой вершины означает ответ «невозможно» для всего теста, это удобнее делать рекурсивным поиском в глубину, который сразу обрывается, если в какой-то вершине разбивать не вышло. Этот поиск в глубину ходит по дереву, в графе, который был до удалений.

Как же осуществить разбиение, если мы сейчас находимся в вершине v? Во-первых рекурсивно разбиваем детей, и если там не вышло разбить (даже с учётом того, что мы разрешаем заимствовать внешнее ребро), то сразу выходим. Будем считать, что рекурсивный обход возвращает нам помимо значения динамического программирования также тип ребёнка — одно из чисел `0`, `+1`, и `-1`, реализующее описанный выше тип пересечения границы поддерева ребёнка.

Заметим, что у детей `0` всё уже хорошо, с ними ничего делать не надо. Рассмотрим мультимножество, в которое поместим все типы детей, отличные от `0`. Заметим, что их можно объединять в пары, компенсируя недостачу входящего или выходящего ребра. Также в мульимножество следует добавить `+1` или `-1` на все рёбра между p и v; они также могут участвовать в объединении; знак числа зависит от ориентации ребра (ведёт в v или из v).

После того, как мы образем максимальное количество пар (`+1`, `-1`), возможны следующие варианты:

- Все объекты знаков `+1` и `-1` объединились в пары. В таком случае мы являемся вершиной типа `0`.
- У нас остался один объект, предположим без ограничения общности, что его знак — `+1`. Заметим, что у нас не было никаких ограничений на то, кого с кем объединять в пары; значит, среди объектов `+1` можно оставить произвольный. Сделаем, так чтобы осталось как раз ребро в предка $v \to p$; тогда в рекурсивном процессе наш предок сможет его объединить в пару с кем-то ещё. Если если же такого ребра не было (удалено или мы корень), то ответ — «невозможно». Действительно, единственный способ добить это ребро до пары — это с помощью ребра вне нашего поддерева, но чтобы иметь возможность образования такой пары, нескомпенсированный `+1` обязан быть ребром в предка.
- У нас осталось более одного объекта одного знака. Тогда тоже ничего сделать нельзя.

Также заметим, что знак действительно определяется однозначно, если мы знаем знаки всех детей и ситуацию с существованием рёбер наверх.

Если неоптимально реализовать процедуру слияния, можно получить решение, которое пройдёт первые несколько групп. Эту же процедуру несложно реализовать и за линейное время от степени вершины; в таком случае решение пройдёт на полный балл.

Разбор задачи «Вырубка деревьев»

Автор и разработчик: Алексей Упирвицкий, Алексей Перевышин

Тривиальные подгруппы

Для начала давайте заметим, что если дерево можно повалить, то это всегда выгодно сделать, так как это не может помешать валить другие деревья. Таким образом, приходим к следующему решению: пока на отрезке есть дерево, которое в данный момент можно повалить, валим его и продолжаем алгоритм. Тривиальная реализация будет работать за $O(n^2)$ на запрос, более аккуратная реализация, которая поддерживает множество деревьев, которые сейчас можно повалить, будет работать за $O(n)$ на запрос.

Альтернативное решение за линейное время

Чтобы получить более быстрое решение, давайте для начала попробуем пересчитывать ответ при добавлении одного дерева к текущему отрезку. Пусть у нас есть некоторый отрезок $l..r$, и мы повалили все деревья, которые смогли. Теперь разрешим валить также дерево $r+1$. Если это дерево нельзя повалить, то ответ остается таким же. Если его можно повалить, то валим его и далее жадно валим все, что можно. Очевидно, что нужно проверять только самое правое из оставшихся деревьев, поэтому это можно делать быстро, храня стек неповаленных деревьев; легко видеть, что средняя амортизационная сложность добавления одного дерева справа составляет $O(1)$, так как каждое дерево будет добавлено и удалено не более одного раза.

Используя описанный метод, можно построить еще одно решение, которое работает за $O(n)$ на запрос, однако его также можно использовать для построения более быстрых решений.

Алгоритм МО

Чтобы решить задачу за время $O((n+q)\sqrt{n})$ воспользуемся алгоритмом МО.

Рассмотрим все запросы, чья левая граница попадает в полуинтервал $[l_i, r_i)$. Заметим, что мы не можем тривиально удалять дерево из рассматриваемого отрезка. Поэтому сделаем следующее: исходно поставим $l = r = r_i$ и будем двигать правый указатель до нужной точки, поддерживая дек неповаленных деревьев. Теперь, когда нужно обработать запрос и правая рассматриваемая граница совпадает с его правой границей, мы должны по 1 добавлять деревья слева. Заметим, что делать это наивно нельзя; каждое добавляемое слева дерево может привести к падению вплоть до всех деревьев, а аргумент про амортизационную стоимость более не применим, т.к. отдельное «дорогое» для добавления дерево может быть многократно обработано в ходе работы алгоритма МО. Для эффективного добавления дерева слева нужно придумать некий способ сжатого представления деревьев.

Что это значит? Когда мы добавляем дерево слева, старые деревья могут упасть только налево. Так как деревьев в блоке всего $O(\sqrt{n})$ - их можно обрабатывать за линию. Теперь мы хотим узнать, сколько деревьев из отрезка $[r_i, r]$ упадут. Заметим, что дерево x упадет если ему будет достаточно расстояния и упадет предыдущее неповаленное. Таким образом, для каждого для каждого дерева можно предпосчитать, какое самое правое дерево слева должно упасть, чтобы мы могло упасть налево. Когда мы идем стеком нужно заменять это значение на минимум из него и значения для предыдущего элемента в стеке. Таким образом в стеке будет невозрастающая последовательность - какое дерево должно упасть чтобы мы упали. Тогда при обработке левого блока можно делать одно из двух: бинарный поиск по этим значениям, что добавит еще $O(\log(n))$ в ассимптотику, или хранить эти значения сжато.

Действительно, нас интересует лишь то, сколько деревьев с каким значением. Можно завести один массив, где добавление дерева это $+1$, а удаление -1. Тогда мы можем просто посмотреть в соответствующую ячейку и узнать, сколько деревьев упадет, если уронить все деревья с полуинтервала $[x, r_i)$.

Divide and Conquer

Чтобы решить задачу за время $O(\log n)$ на запрос, воспользуемся следующей идеей.

Рассмотрим некоторое дерево. Если его можно повалить, то его можно повалить либо налево, либо направо. Пусть его можно повалить налево. Тогда деревья справа от него этому не мешают.

Кроме этого, добавление новых деревьев слева также этому не помешает. Таким образом, есть некоторое число a_i такое, что дерево i можно повалить налево если и только если левая граница отрезка $l \leqslant a_i$. Аналогично, есть некоторое число b_i такое, что дерево i можно повалить направо если и только если правая граница отрезка $r \geqslant b_i$.

Предположим, что мы вычислили числа a_i и b_i. Тогда задача сводится к следующей: для каждого запроса (l, r) требуется найти число деревьев, для которых $l \leqslant a_i$ и $r \geqslant i$ или $l \leqslant i$ и $r \geqslant b_i$. Это можно сделать с помощью стандартной техники с помощью сканирующей прямой и дерева отрезков.

Осталось научиться вычислять числа a_i и b_i. Покажем, как вычислить числа a_i, числа b_i вычисляются аналогично. Будем вычислять числа a_i с помощью двоичного поиска.

В каждый момент времени у нас будет некоторый полуинтервал $[L..R)$ и множество деревьев, и мы знаем, что для всех деревьев в этом множестве число a_i находится в полуинтервале $[L..R)$. Выберем значение M, среднее между L и R. Запустим алгоритм со стеком, начиная с позиции M. Если дерево не удалось повалить налево, то для него $a_i < M$, если удалось, то $a_i \geqslant M$. Отнесем такие деревья к множествам A и B, соответственно.

Рассмотрим деревья из множества A. Для них нужно запустить двоичный поиск на полуинтервале $[L..M)$, при этом заметим, что деревья из множества B для любого значения из этого полуинтервала точно упадут, поэтому они не будут мешать деревьям из множества A. Таким образом, в этом рекурсивном вызове можно считать, что деревьев из множества B не существует.

Аналогично для деревьев из множества B нужно запустить двоичный поиск на полуинтервале $[M..R)$, при этом деревья из множества A не могут помешать им упасть, так как они не помешали им упасть для значения M. Таким образом, в этом рекурсивном вызове можно считать, что деревьев из множества A не существует.

Итого мы получили рекурсивный алгоритм, который каждый раз делит диапазон значений пополам, а деревья на два непересекающихся множества. Время работы такого алгоритма $O(n \log n)$, так как глубина рекурсии равна $\log n$, и общее число элементов на каждом слое рекурсии равно n.

Итоговое время работы алгоритма $O((n + q) \log n)$.

Рассмотрим альтернативный способ вычисления чисел a_i. Будем вычислять их в порядке возрастания i. Для фиксированного i изначально установим значение $a_i = i$ и будем повторять следующие действия:

1. Если дерево $a_i - 1$ не мешает дереву i упасть влево, значение a_i найдено правильно, выходим. В противном случае значение a_i точно нужно уменьшить.

2. Если дерево $a_i - 1$ может упасть вправо, не задев дерево i, уменьшим значение a_i на единицу и вернёмся к пункту 1.

3. Иначе заметим, что дерево $a_i - 1$ придётся повалить влево, а вместе с ним и все деревья от a_{a_i-1} до $a_i - 2$. В таком случае установим $a_i = a_{a_i-1}$ и также вернёмся к пункту 1.

Чтобы оценить асимптотику этого способа вычисления a_i, рассмотрим некоторое дерево j. Пусть $i_1, i_2, \ldots, i_k$ $(j < i_1 < i_2 < \ldots < i_k)$ — это номера деревьев, для которых в некоторый момент значение a_i было равно $j + 1$, но в итоге оказалось меньше.

Заметим, что если в некоторый момент $a_i = j + 1$, то все деревья от $j + 1$ до $i - 1$ включительно можно повалить либо влево, либо вправо, не задевая деревья j и i.

Рассмотрим любой индекс t $(1 \leqslant t < k)$. Из определения i_t мы знаем, что дерево i_t нельзя повалить влево, не задевая дерево j. Следовательно, его должно быть можно повалить вправо, не задевая дерево i_{t+1}.

Значит, расстояние между деревьями j и i_t не превосходит расстояния между деревьями i_t и i_{t+1}. Или, что эквивалентно, расстояние между деревьями j и i_t хотя бы вдвое меньше расстояния между деревьями j и i_{t+1}. Значит, $k = O(\log C)$, где C — максимальная координата дерева, а приведённый алгоритм вычисления a_i работает за $O(n \log C)$.

Разбор задачи «Блогеры-путешественники»

Автор: Ильдар Гайнуллин
Разработчик: Иван Сафонов

В первой подзадаче данный граф был деревом. Поэтому его можно обойти с помощью dfs и посчитать ответ для каждой вершины (так как путь от вершины 1 до нее единственный).

Для того, чтобы решить четвертую и пятую подзадачи, нужно организовать перебор всех путей из вершины 1. В четвертой подзадаче достаточно перебрать все перестановки ребер и проверить все префиксы этих перестановок. Такое решение работает за $O(m!m)$. Для того, чтобы пройти пятую подзадачу, нужно было реализовать любой перебор всех реберно-простых путей с помощью функции перебора. Утверждается, что если в графе **нет кратных ребер** и количества вершин и ребер $\leqslant 20$, то количество реберно-простых путей крайне мало.

В шестой подзадаче можно заметить, что любой интересный реберно-простой путь будет выглядеть так: [путь от 1 до j] + [путь от j до i] ($i \leqslant j$). Первая часть пути при этом проходит по минимальным ребрам между парами вершин p и $p+1$, вторая часть пути проходит по вторым минимальным ребрам между парами вершин p и $p+1$ (чтобы путь был, они должны существовать для всех $i \leqslant p < j$). Можно перебрать все $O(n^2)$ этих путей и обновить ответы для всех вершин.

Во второй подзадаче надо заметить, что цена минимального ребра для любого пути из вершины 1 будет равна 0. Значит нам нужно найти путь с минимально возможной максимальной ценой ребра на пути. Это известная задача, ее решение состоит в том, что надо найти минимальное остовное дерево и взять пути из него.

В третьей подзадаче надо заметить, что цена максимального ребра для любого пути из вершины 1 будет равна 10^9. Значит нам нужно найти путь с минимально возможной минимальной ценой ребра на пути. Построим компоненты вершинной двусвязности и дерево на них. Известно, что ребра этого дерева это мосты графа. Любой реберно-простой путь из вершины 1 в вершину p будет выглядеть как путь внутри этого дерева из компоненты вершины 1 в компоненту вершины p, а внутри компонент будет выбран какой-то путь между нужными концами. Можно доказать, что в любой компоненте вершинной двусвязности можно выбрать реберно-простой путь между заданной парой вершин, проходящий по заданному ребру. Значит внутри одной компоненты можно будет выбрать реберно-простой путь, проходящий по минимальному ребру внутри этой компоненты. Поэтому найдем в каждой компоненте минимальную цену ребра. Ответ для вершины p будет равен минимальной цене ребра по всем компонентам и ребрам на пути от компоненты вершины 1 до компоненты вершины p.

Решение третьей подзадачи очень поможет нам решить задачу в общем случае. Зафиксируем вес максимального ребра и скажем, что он будет $\leqslant t$. Тогда оставим только ребра i, такие что $c_i \leqslant t$. Среди таких ребер нам нужно для каждой вершины p найти минимально возможную цену минимального ребра на пути — f_p. Мы можем сделать это за $O(m)$ (как в шестой подзадаче). Далее обновим для каждой вершины p ответ числом $t + f_p$. Легко понять, что для каждой вершины мы найдем верный ответ, потому что каждый путь будет учтен в тот момент, когда мы рассмотрим t равным максимальной цене ребра на этом пути.

В седьмой подзадаче нужно перебрать значение t равное каждой цене ребра. Время работы решения $O(m^2)$.

В восьмой подзадаче надо заметить, что нужно оставить только такие ребра, добавление которых меняет компоненты вершинной двусвязности (если мы будем добавлять ребра в порядке возрастания цены). Легко понять, что количество таких ребер $\leqslant 2n$ (не нужно рассматривать ребро, которое при добавлении соединяет две вершины из одной компоненты вершинной двусвязности). Если оставить только такие ребра, то предыдущее решение будет работать за $O(n^2 + m \log m)$.

Далее опишем полное решение задачи.

Построим минимальное остовное дерево T. Будем идти по ребрам в порядке возрастания цены. Ограничение t на максимальную цену ребра на пути будет равно цене текущего ребра. Нам нужно научиться как-то быстро обновлять ответы для вершин числами $t + f_p$.

Будем поддерживать в СНМ на множестве вершин текущие компоненты реберной двусвязности. Заметим, что каждая компонента реберной двусвязности всегда будет являться связным поддеревом T.

Когда мы добавляем новое ребро (s, f) есть несколько случаев:

1. s и f лежат в одной компоненте реберной двусвязности. В этом случае это ребро можно пропустить, так как оно не обновит ответы.

2. Ребро (s, f) лежит в T. В этом случае компоненты реберной двусвязности не поменяются. Единственное, что может поменяться — некоторые новые вершины могут стать достижимы из вершины 1.

3. Ребро (s, f) не лежит в T, но s и f из разных компонент реберной двусвязности. В этом случае компоненты реберной двусвязности на пути между s и f объединяются в одну.

Поймем, как в случаях 2 и 3 нужно обновлять ответы. Пусть, не умаляя общности, глубина вершины s не больше глубины вершины f.

В случае 2 обойдем вершины, которые станут достижимы из вершины 1 (а до этого не были). Это можно сделать обходом в глубину из вершины f (проходя по ребрам T вес которых $\leqslant t$). Для каждой из этих вершин ответ нужно установить равным f_p+t, где f_p равно текущему минимальному ребру на пути от 1 до p (включая ребра внутри компонент реберной двусвязности).

В случае 3 объединим в СНМ компоненты реберной двусвязности на пути между s и f. Это можно сделать за $O(\alpha(n)\ell)$, где ℓ — длина этого пути. Далее для всех вершин p, таких что на пути от 1 до p лежит новая компонента, нужно обновить ответ числом w_s+t, где w_s — цена минимального ребра внутри этой компоненты.

У нас есть несколько нерешенных вопросов:

1. Как поддеживать числа w_s для всех компонент реберной двусвязности?

2. Как поддерживать числа f_p для всех вершин?

3. Как обновлять ответы быстро?

Для решения первого вопроса просто будем поддерживать эти числа для каждой компоненты в СНМ и обновлять при объединении компонент. Также для каждой компоненты будем поддерживать самую высокую вершину из этой компоненты (вершину с минимальным расстоянием до 1 в дереве T).

Для решения второго вопроса будем поддерживать дерево отрезков на массиве вершин, расположенных в порядке эйлерова обхода дерева T. Его значениями будут f_p. Тогда:

- В случае 2 нужно обновить числа в поддереве T для вершины f числом t. Поскольку вершины поддерева это отрезок эйлерова обхода, нам нужно сделать групповую операцию $min =$ на отрезке.

- В случае 3 рассмотрим самую высокую вершину v новой объединенной компоненты. Тогда нам нужно обновить числа в поддереве T для вершины v числом t (потому что если эта компонента лежит на пути от 1 до p, то вершина v тоже на нем лежит). Для этого аналогично нужно сделать групповую операцию $min =$ на отрезке.

Для решения третьего вопроса аналогично будем поддерживать другое дерево отрезков на массиве вершин, расположенных в порядке эйлерова обхода дерева T. Его значениями будут ответы для вершин. Тогда:

- В случае 2 нужно в этом ДО выставлять конкретное значение $f_p + t$ в позиции вершины p (не смотря на посчитанный ответ до этого).

- В случае 3 рассмотрим самую высокую вершину v новой объединенной компоненты. Тогда нам нужно обновить ответ в поддереве T для вершины v числом $w_s + t$. Для этого нужно сделать групповую операцию $min =$ на отрезке. Заметим, что мы можем обновить ответ для некоторых вершин, которые сейчас не достижимы из 1, но в этом случае мы эти ответы все равно забудем, когда будем выставлять правильное значение в тот момент, когда эта вершина станет достижима.

Получаем, что для решения задачи нам нужно дерево отрезков с операцией «$min = x$» (уменьшить значение до x, если оно больше x) на отрезке, выставление конкретного значения в точке, получение конкретного значения в точке.

В итоге мы получаем решение за $O(m \log n)$.