Задача 1. Распределённые системы

Краткое условие задачи:

Имеется n серверов, на i-м сервере запущено a_i служб. Для каждого сервера i задан резервный сервер p_i; массив p — перестановка, и $p_i = i$ только для «надёжных» серверов, которые никогда не отключаются. Если сервер i отключается, его службы мгновенно перемещаются на p_i, а сам сервер заменяется пустым (его номер и резервный сервер сохраняются). Отключения происходят последовательно, их количество не превышает k. Нужно определить максимальное возможное число служб, которые могут оказаться одновременно на одном сервере после отключения не более k

Краткая идея решения:

Поскольку p — перестановка, граф, образованный ребрами $i \to p_i$, состоит из независимых ориентированных-циклов. Отключая серверы, мы «склеиваем» их со следующим по направлению ребра, так что после отключения не более k серверов все службы могут оказаться на одном сервере только из последовательных (по направлению цикла) максимум $k+1$ серверов.

Следовательно, ответ равен максимальной сумме a_i по любому (циклическому) подотрезку длины $L = min|C|, k+1$, где $|C|$ — длина рассматриваемого цикла.

Для каждого цикла считаем максимум суммы L последовательных элементов (все $a_i \geqslant 0$, поэтому более короткий отрезок не даёт большего результата). Это делается скользящим окном за $O(|C|)$. Суммируя время выполнения по всем циклам получаем решение за $O(n)$ времени и $O(n)$ памяти.

Подробный разбор:

Подзадача 1 ($k = 1$)

При отключении одного сервера i его службы переходят на p_i. Если $p_i = i$ (надёжный сервер), то его нагрузка останется равной a_i независимо от отключения. Если $p_i \neq i$, то после отключения i на сервере p_i будет $a_{p_i} + a_i$ служб.

Мы можем также решить не отключать ни одного сервера, тогда нагрузка остаётся a_j на каждом сервере j.

Алгоритм:

- Инициализировать ответ $ans = max_j a_j$ (случай без отключений).
- Для каждого i с $p_i \neq i$ обновить $ans = max(ans, a_i + a_{p_i})$.

Сложность — $O(n)$ времени, $O(1)$ памяти.

Подзадача 2 ($n \leqslant 1000$)

Теперь k может быть произвольным, но n маленькое, поэтому сложность может быть не оптимальной.

Поскольку граф — набор независимых ориентированных-циклов. Отключения в разных циклах не влияют друг на друга, потому что службы никогда переходят из одного цикла в другой. Поэтому задачу можно решить по отдельности для каждого цикла и взять максимум.

Организуем перебор внутри цикла. Пусть цикл содержит m серверов. Запишем их количества служб в массив $b[0 \ldots m-1]$ в порядке следования по ребрам $i \to p_i$. После отключения t ($0 \leqslant t \leqslant k$) серверов, расположенных подряд, их службы соберутся на следующем сервере, т.е. там окажется сумма на отрезке длины $t+1$.

Следовательно, ответ для данного цикла — максимальная сумма элементов на циклическом подотрезке длиной не более $k+1$.

Будем перебирать начала всех подотрезков. Для каждого начала l (от 0 до $m-1$) перебираем $len = 1 \ldots min(m, k+1)$ и считаем сумму $b[l] + b[l+1] + \ldots + b[l+len-1]$ (индексы берём по модулю mm). Сохраняем глобальный максимум.

Суммарная сложность $O(\sum(|C|^2)) \leqslant O(n^2)$ что укладывается в ограничение.

Подзадача 3 ($p_i = i \bmod n + 1$) , один цикл)

Задан один цикл длины n. Нужно найти максимум суммы на циклическом подотрезке длиной не более L, где $L = min(n, k+1)$.

Так как $a_i \geqslant 0$, удлинение подотрезка никогда не уменьшает сумму. Поэтому оптимальный подотрезок будет самой большой возможной длины L

Реализуем алгоритм скользящего окна. Сформируем массив b длиной $2n$, копируя исходный массив a дважды подряд (это позволяет пройти по циклу без модулей). Вычислим сумму первых L элементов: $cur = \sum_{i=0}^{L-1} b[i]$. Инициализируем $ans = cur$. Для i от 1 до $n-1$ реализуем сдвиг окна $cur = cur - b[i-1] + b[i+L-1]$ (сдвиг окна на один вправо). На кажодм шаге выбираем $ans = max(ans, cur)$.

Сложность — $O(n)$ времени, $O(n)$ памяти (можно и $O(1)$, если использовать кольцевой буфер).

Полное решение

Разложим граф на циклы. Создадим массив $used[1..n]$, в котором будет храниться, посещен ли узел или нет. Будем идти по ребрам пока не вернёмся к уже посещённому узлу, собирая все вершины в текущий цикл.

Для найденного цикла определим максимальную сумму как в предыдущей подзадаче. Суммарная длина циклов равна n, таким образом сложность правильного решения равна $O(n)$.

Задача 2. Расследование в Темерии

Краткое условие

Дано дерево с n вершинами, корень — вершина 1, ребро i имеет длину w_i. Трисс должна начать и закончить путь в корне, посетив все вершины. У неё есть не более k кристаллов телепортации. В любой момент она может оставить кристалл в текущей вершине и позже мгновенно переместиться к любой вершине, где уже был оставлен кристалл, проходя по единственному пути между ними. После использования кристалла все вершины этого пути «запятнаны» и больше не могут участвовать в телепортах.

Для каждого $j = 1 \dots k$ найдите минимальное пройденное расстояние, если можно использовать не более j кристаллов.

Замечание

Заметим, что эффективно телепортироваться только по вертикальным путям (то есть из вершины v телепортироваться в какого-то своего предка). Тогда задача сводится к тому, чтобы найти набор из максимальных по весу k вершинно-непересекающихся путей.

Подзадача 1: $n \leqslant 150\,000$; $k = 1$

В этой подзадаче достаточно было найти самый большой вертикальный путь — его величина равна высоте дерева.

Подзадачи 2–4: $n \leqslant 5\,000$ или $n \leqslant 150\,000$; $k \leqslant 300$

Воспользуемся методом динамического программирования: вычислим значения $\mathrm{dp}[v][s]$ — максимальная суммарная стоимость s непересекающихся по вершинам путей в поддереве v. Так как все веса положительны, то это означает, что в таком разбиении всегда будет путь, начинающийся в вершине v, потому что можно взять путь, начинающийся в ближайшей к v вершине, и продлить его.

Если вычислять данную $\mathrm{dp}[v][s]$ для всех $v \in [1, n]$ и для всех $s \in [0, k]$, то наивно можно оценить вычисление этой ДП временем работы $O(nk^2)$. Но эта ДП — частный случай рюкзаков на дереве, а поэтому, если дополнительно ограничить измерение s размером поддерева v сверху, это позволяет получить лучшую оценку на время работы.

Одной из самых известных оценок на время работы такой ДП является $O(n^2)$. Действительно, время работы можно оценить как сумму $\sum_v \prod_{u_1 \neq u_2} \mathrm{sz}(u_1)\mathrm{sz}(u_2)$, где суммирование ведется по всем вершинам v, а u_1 и u_2 — непосредственные потомки вершины v, и $\mathrm{sz}(w)$ обозначает количество вершин в поддереве w.

С одной стороны, эта сумма подсчитывает суммарное количество пар вершин, находящихся в различных поддеревьях относительно какой-то вершины v, а с другой стороны, каждая пара вершин входит в эту сумму не более одного раза, откуда и следует искомая оценка.

Менее известной является оценка времени работы $O(nk)$. Ее можно получить, если отдельно оценить сверху объединения пар двух ДП из поддеревьев размером больше k, объединение всех пар поддеревьев не больше k и, наконец, объединение пар поддеревьев, где одно поддерево не больше k, а другое — больше k.

Подзадачи 5–6: $n \leqslant 150\,000$, $w_i = 1$

Если $w_i = 1$, то работает жадный алгоритм — достаточно каждый раз выбирать самый длинный путь, который остался в дереве, и добавлять его в ответ. Такой алгоритм можно эффективно реализовать следующим образом:

- Для каждого поддерева v явно вычислим его высоту $h(v)$;
- Проведем тяжелое ребро из вершины v в самое высокое поддерево этой вершины. Если таких деревьев несколько, то проведем тяжелое ребро в одно из этих поддеревьев;
- В конце мы получили явное разбиение дерева на тяжелые пути, и ясно, что существует вариант исполнения жадного алгоритма, когда будут выбраны именно такие пути.
- Остается только выделить эти пути, отсортировать их по убыванию их длины и насчитать префиксные суммы на массиве их длин — это и будет искомый ответ на задачу.

Подзадача 7: $n \leqslant 150\,000$, специальный граф

Пусть у вершины 1 есть m детей, которые состоят из ребер с весами x_i и y_i в порядке сверху-вниз. То есть ребро x_i из i-го пути инцидентно корню дерева, а ребро с весом y_i инцидентно листу из i-го пути. Тогда ясно, что ответ выглядит как:

$$\text{ans}(k) = \max_{B \subset 1,2,\ldots,m, |B|=k} \left\{ \max_{i \in B} x_i + \sum_{i \in B} y_i \right\}$$

Это так, потому что, если нам надо выделить k путей, то надо сначала выбрать, в каких поддеревьях будут находиться эти пути, и ровно один из этих путей сможет быть продлен до корневой вершины.

Чтобы найти искомую стоимость разбиения на k, отсортируем все пары (x_i, y_i) по возрастанию x_i. В ходе алгоритма мы эти пары разделим на два типа: те пары, где x_i выступает максимумом в каком-то оптимальном выборе, и где x_i никогда не становится максимальным.

Заметим, что задачу можно решить жадно, добавляя пути по одному. Пути можно добавлять по одному, для этого достаточно рассматривать либо пару, где максимален x_i, либо путь, в котором максимально значение $x_i + y_i$ (с предположением, что максимум увеличится). Это можно сделать, заранее отсортировав массив двумя способами, и выбирать очередной максимум. Если какой-то элемент не подходит под жадное изменение, то его можно просто игнорировать.

Общие замечания про жадные алгоритмы.

В этой задаче в общем случае работает жадный алгоритм, который является комбинацией идей из подгрупп 6 и 7. Можно добавлять пути по очереди двумя способами:

- Очередной путь добавляется по еще не задействованным вершинам в других путях;
- Очередной путь проходит по одной задействованной вершине и идет вниз по незадействованным вершинам. При этом одно ранее добавленное ребро удаляется.

Доказать этот жадный алгоритм можно с помощью MCMF (mincost maxflow): раздвоим каждую вершину дерева i на две вершины u_i и v_i и добавим две фиктивные вершины s и t, ребра устроены следующим образом:

- Есть ребра из s в u_i для всех i с нулевой стоимостью;
- Есть ребра из v_i в t для всех i с нулевой стоимостью;
- Есть ребра из u_i в v_i для всех i с нулевой стоимостью;
- Для каждого вертикального ребра $j = p_i$ есть ребро в сети из v_j в u_i со стоимостью веса исходного ребра дерева.

Все пропускные способности равны единице. Если наивно рассмотреть всевозможные увеличивающие цепочки в этой сети (с требованием на простоту пути в остаточной сети), то как раз и получится два жадных способа увеличения количества путей.

Подзадачи 8–11. $n \leqslant 500\,000$

С помощью структур данных (например, дерево отрезков в комбинации с эйлеровым обходом дерева)можно жадно поддерживать стоимость добавления пути первым и вторым способом (описанных в подгруппе ранее). При этом суммарная длина путей не будет превосходить $n + k$ по очевидным причинам, поэтому каждое добавленное ребро можно учитывать отдельно.

Другой подход заключается в том, чтобы объединить решение подзадач 6–7 и совместить жадный алгоритм с динамическим программированием. Для каждой вершины будем поддерживать массив изменений суммарной стоимости путей, если количество путей увеличивать на один. Чтобы объединить ДП, достаточно просто объединить все изменения, идущие после первого добавленного пути. А чтобы понять, какие первые добавленные пути в каждом поддереве вершины v, требуется игнорировать дальнейшие изменения и воспользоваться подзадачей 7.

Задача 3. Скобки и деревья

Краткое условие задачи

В первом запуске интерактивной программы подается n правильных скобочных последовательностей $s_1, \ldots, s_n$ — коды неупорядоченных корневых деревьев. Нужно вывести одну ПСП w, кодирующую некоторое дерево, из которого во втором запуске (получив любую ПСП, кодирующую то же дерево) можно восстановить исходные $s_1, \ldots, s_n$ в том же порядке. Ограничения — длина w должна быть $O(\sum |s_i|)$, конкретные ограничения на длину указаны в подгруппах.

Подгруппы 1–2: $f(x) = x + 2000$, $S \leqslant 200\,000$, При втором запуске даются ПСП, точно совпадающие с выведенными при первом запуске или $t_1 < t_2 < \ldots < t_n$

В этих подгруппах можно закодировать данные строки $s_1, \ldots, s_n$ как «$(s_1 \ldots s_n)$». Этот способ не работает в общем случае, поскольку, хотя порядок детей в деревьях неважен, нам нужно будет восстановить порядок самих деревьев.

Во второй подгруппе размеры деревьев строго возрастают, поэтому мы можем восстановить исходный порядок, отсортировав по возрастанию длины строки $s'_1, \ldots, s'_n$ из полученной ПСП «$(s'_1 \ldots s'_n)$».

Реализация этих групп может работать только со строками, то есть не строить деревья из ПСП.

Подгруппа 3: $f(x) = x + 2000$, $S \leqslant 200\,000$, $n = 2$

В этой подгруппе нам нужно передать единственный бит информации, отвечающий за порядок двух деревьев. Это можно сделать, например, за $m = s + 8$, таким образом:

- если $|s_1| \leqslant |s_2|$, выведем «(((s_1)(s_2))»;
- если $|s_1| > |s_2|$, выведем «(((s_1)(s_2)())».

s_1 и s_2 можно получить как два поддерева на глубине 2, а их порядок восстанавливается в зависимости от количества детей у корня.

Подгруппа 4: $f(x) = 4 \cdot x + 2000$, $S \leqslant 200\,000$, $n = 2$

В этой подгруппе в силу неточных ограничений возможны многие решения. Наиболее простые из них используют авторскую идею: построим бамбук высоты n и будем подвешивать i-е дерево к его вершине на высоте i (возможно, через какие-то дополнительные вершины).

Например, решение с ПСП длины $m = s + 4n + 6$ может выглядеть так:

$$((s_1)((s_2)(\ldots((s_n)(()()))\ldots)))$$

Здесь мы подвешиваем к концу бамбука поддерево «(()())»; нам нужно отличить наш бамбук от других путей, заканчивающихся этим поддеревом. Для этого встанем в корень и каждый раз будем спускаться в вершину с двумя детьми (за вершину с одним ребёнком подвешено s_i).

Поскольку $s \geqslant 2n$, $m = s + 4n + 6 \leqslant 3s + 6$, причём оценка достигается только на тесте из n деревьев размера 1. Поэтому, чтобы улучшить константу при s, нужно более аккуратно работать с деревьями маленького размера. Например, если i-е дерево — бамбук, мы можем подвешивать его напрямую к бамбуку, а не через промежуточную вершину.

Другое решение строит бамбук большей длины, чтобы его конец был самой глубокой вершиной в дереве; для этого достаточно $n + \frac{1}{2}s$ вершин.

Подгруппа 5: $f(x) = x + 2000$, $t_1 = t_2 = \ldots = t_n > 1$

В этой подгруппе будем развивать конструкцию из предыдущей подгруппы, но избавимся от дополнительного бамбука. Вместо добавления отдельных вершин расположим корни исходных деревьев в один путь. То есть подвесим корень второго дерева как ребенка корня первого дерева. Корень третьего дерева подвесим как ребенка корня второго дерева и так далее. Так как размеры всех деревьев одинаковы, то подвешенный корень всегда будет самым большим поддеревом у корня предыдущего дерева.

Заметим, что тогда в случае, если исходно было больше одного дерева, то можно легко восстановить все деревья — у корня возьмем все поддеревья, кроме самого большого по размеру, это дает нам первое дерево. Дальше спускаемся в самого глубокого ребенка, пока его размер не станет равным размеру первого дерева.

Чтобы обработать случай одного дерева, подвесим получившийся путь корней деревьев к отдельной вершине. К этой вершине подвесим отдельный лист в случае, если исходное дерево было одно. Так по количеству детей корня сможем определить, было ли одно дерево или несколько.

Подгруппа 6: $f(x) = x + 2000$, $t_i > 1$

Для этой подгруппы будем использовать похожую конструкцию, как в предыдущей группе, с путем из корней деревьев. Однако, если построить такой же путь из корней деревьев, то теперь не всегда у очередного корня самым большим ребенком будет корень следующего дерева. Такое может происходить, когда размер текущего дерева больше суммарного размера всех оставшихся деревьев. Назовем такие деревья тяжелыми (в том числе последнее дерево тоже является тяжелым).

Заменим тяжелые деревья следующей конструкцией: Создадим отдельную вершину. К ней подвесим ребенком ещё одну вершину, а уже к ней подвесим корень тяжелого дерева. Теперь создадим путь из корней деревьев, как мы и делали в предыдущей подгруппе, и подвесим к нему в конце одну вершину.

Теперь надо научиться декодировать тяжелые деревья. Заметим, что, когда мы будем спускаться по пути корней в самого тяжелого ребенка, то, когда мы окажемся в тяжелом дереве, мы спустимся в вершину с ровно одним ребенком. Так как у нас нет деревьев единичного размера, то такие вершины не могут лежать на пути из корней деревьев. Таким образом, мы сможем определять тяжелые деревья и спускаться ниже в другую ветку. Последняя вершина, которую мы отдельно подвесили к концу пути корней, будет индикатором окончания деревьев.

Заметим, что тяжелых деревьев не будет много. Так как тяжелое дерево по размеру больше суммарного размера всех следующих деревьев, то каждое тяжелое дерево хотя-бы в 2 раза увеличивает суммарный размер всех деревьев. Поэтому количество добавленных вершин будет не более удвоенного логарифма количества деревьев.

Полное решение

Альтернативный способ пометить бамбук, за который мы подвешиваем данные деревья — это подвесить за его конец случайно сгенерированный «маркер». Тогда на втором запуске можно будет найти его с помощью хеширования поддеревьев.

Решения с константным маркером не работают, потому что интерактор может подавать на вход решению части деревьев, выведенных участником ранее.

Один из способов обойти это — инициализировать случайный генератор некоторой характеристикой входа. Например, можно генерировать маркер фиксированного размера и инициализировать генератор размером ввода, поскольку он будет известен на обоих запусках.

Решение на 100 баллов должно использовать маркер размера не больше 15, но количество корневых деревьев размера 15 (87 811) уже сравнимо с ограничениями задачи ($\lfloor 10^6/30 \rfloor = 33\,333$). Поэтому возможен тест, который содержит много различных поддеревьев размера 15 и с большой вероятностью содержит сгенерированный маркер.

Мы можем представить себе битовую маску длины 87 811, в которой i-й бит включён, если вход содержит i-е дерево размера 15 (например, в порядке возрастания хеша). Когда мы выбираем маркер, мы дополнительно включаем ещё один бит, который мы должны восстановить на втором запуске.

Посчитаем префиксные балансы, считая, что включённые биты дают $+1$, а выключенные -1. В конце баланс будет отрицательным; пусть минимальный баланс впервые достигается на i-м бите. Тогда включим его (выберем в качестве маркера i-е дерево); на суффиксе баланс увеличится на 2. После этого изменения новый минимальный баланс в последний раз достигается на $(i-1)$-м бите. Это нам и нужно найти на втором запуске.

Задача 4. Спортивная тренировка

Решение 1

Самая важная мысль состоит в том, что после перехода к порядку по возрастанию значений сами значения больше не важны. Остаётся только строка над алфавитом $\{L, R, M\}$.

Пусть текущие участники стоят в ряд, а pos_x — позиция участника с номером x в этом ряду. Рассмотрим участников в порядке возрастания номеров.

Предположим, что все меньшие номера уже обработаны. Среди них для нас важны только две величины:

- самая левая позиция;
- самая правая позиция.

Для очередного участника x возможны ровно три случая:

- pos_x левее всех меньших — тогда пишем символ `L`;
- pos_x правее всех меньших — тогда пишем символ `R`;
- pos_x находится между ними — тогда пишем символ `M`.

Глобальный минимум не кодируется, так как у него нет меньших элементов.

Таким образом, любой текущий набор участников кодируется строкой

$$s \in \{L, R, M\}^*,$$

записанной **в порядке возрастания номеров**.

После этого задача уже формулируется не про самих школьников, а про то, как по этой строке быстро считать ответ.

Как построить строку $L/R/M$ с нуля

Это нужно и для маленьких подзадач, и для инициализации полного решения.

Пусть текущие участники имеют номера

$$x_1 < x_2 < \cdots < x_k$$

в порядке возрастания, а их позиции в ряду равны

$$p_1, p_2, \ldots, p_k.$$

Тогда:

- x_1 — текущий глобальный минимум, его пропускаем;
- дальше поддерживаем
$$mn = p_1, \qquad mx = p_1;$$
- для каждого $i \geqslant 2$:
  - если $p_i < mn$, пишем `L` и обновляем $mn = p_i$;
  - если $p_i > mx$, пишем `R` и обновляем $mx = p_i$;
  - иначе пишем `M`.

Это обычный линейный проход после того, как известны позиции текущих участников.

Подзадачи

Подзадачи 1 и 10: $n + q \leqslant 16$

Здесь можно вообще не использовать специальную структуру.

После каждого изменения:

1. строим текущий граф допустимых бросков напрямую по определению;
2. ищем в нём максимальный простой путь битмасочным DP:
$$dp[\text{mask}][v] = \text{существует ли путь, использующий mask и заканчивающийся в } v.$$

Сложность одного пересчёта равна

$$O(m^2 2^m),$$

где m — текущее число участников.

Для $m \leqslant 16$ этого более чем достаточно.

Подзадачи 2, 4, 11, 13: $n, q \leqslant 1000$

Здесь уже можно использовать кодирование строкой $L/R/M$, но пока без сложных структур.

После каждого запроса делаем всё заново:

1. восстанавливаем текущий порядок участников;
2. строим строку s за $O(m)$;
3. считаем ответ по этой строке:
   - для $t = 1$ — линейным DP;
   - для $t = 2$ — прогоном по маленькому автомату.

Итого получается

$$O(m)$$

на один запрос и

$$O((n+q)^2)$$

суммарно.

Этого достаточно для ограничений порядка $n, q \leqslant 1000$.

Подзадачи 3, 5, 12, 14: $q = 0$

Здесь строку нужно построить только один раз и только один раз по ней посчитать ответ.

Сложность:

$$O(n).$$

Подзадачи 6 и 15: $n = 1$, $a_1 = 1$, добавления по возрастанию номеров

В этом частном случае каждый новый участник является новым максимумом по номеру. Значит, относительно всех меньших он может оказаться только:

- либо слева от всех — символ `L`;
- либо справа от всех — символ `R`.

Символов `M` не бывает, а старые символы не меняются.

То есть строка просто растёт справа, и на каждом запросе к ней дописывается одна буква `L` или `R`.

Дальше:

- для $t = 1$ обновляем несколько чисел за $O(1)$;
- для $t = 2$ делаем один переход автомата за $O(1)$.

Подзадачи 7, 8, 9, 16, 17, 18

Здесь уже требуется полное решение.

Полное решение для $t = 1$

Что нужно хранить

Пусть уже обработан некоторый префикс строки s. Для будущих добавлений важны только две крайние вершины среди уже меньших:

- левая крайняя;
- правая крайняя.

Поэтому достаточно хранить три величины:

- $f(s)$ — длина максимального пути;
- $dp_L(s)$ — длина максимального пути, у которого один конец — левая крайняя вершина;
- $dp_R(s)$ — длина максимального пути, у которого один конец — правая крайняя вершина.

Это и есть всё состояние.

Почему этого хватает

При добавлении одной новой вершины происходит ровно одно из трёх:

- `L`: новая вершина соединяется только с правой крайней;
- `R`: новая вершина соединяется только с левой крайней;
- `M`: новая вершина соединяется с обеими крайними.

Значит, новый лучший путь можно получить только так:

- оставить старый лучший путь;
- продлить путь, который заканчивался в левой границе;
- продлить путь, который заканчивался в правой границе.

Никакая другая информация о старом графе для будущего не нужна.

Переходы по одной букве

Символ `L`. Новая вершина становится новой левой границей и соединяется с прежней правой:

$$\texttt{L}: \begin{cases} f' = \max(f,\ dp_R + 1), \\ dp'_L = dp_R + 1, \\ dp'_R = dp_R. \end{cases}$$

Символ `R`. Симметрично:

$$\texttt{R}: \begin{cases} f' = \max(f,\ dp_L + 1), \\ dp'_L = dp_L, \\ dp'_R = dp_L + 1. \end{cases}$$

Символ `M`. Новая вершина цепляется к обеим границам, но сами границы не меняются:

$$\texttt{M}: \begin{cases} f' = \max(f,\ dp_L + 1,\ dp_R + 1), \\ dp'_L = dp_L, \\ dp'_R = dp_R. \end{cases}$$

Для пустой строки:

$$f = 0, \qquad dp_L = 0, \qquad dp_R = 0.$$

Если строку нужно посчитать только один раз, то этого уже достаточно для линейного решения.

Как пересчитывать это в дереве отрезков

Каждый символ действует на тройку (f, dp_L, dp_R) как некоторое преобразование. Поэтому любой отрезок строки можно понимать как преобразование, которое переводит состояние перед этим отрезком в состояние после него.

Композиция двух соседних отрезков — это просто композиция соответствующих преобразований. Именно поэтому в вершине дерева отрезков удобно хранить не сам ответ на отрезке, а преобразование для этого отрезка.

Аналогично случаю с одной буквой, удобно записать состояние в виде столбца

$$v = \begin{pmatrix} f \\ dp_L \\ dp_R \end{pmatrix}.$$

Тогда каждую букву можно записать как max-plus матрицу размера 3×3.

Max-plus матрицы

Для символов `L`, `R`, `M` получаем:

$$A_L = \begin{pmatrix} 0 & -\infty & 1 \\ -\infty & -\infty & 1 \\ -\infty & -\infty & 0 \end{pmatrix}, \qquad A_R = \begin{pmatrix} 0 & 1 & -\infty \\ -\infty & 0 & -\infty \\ -\infty & 1 & -\infty \end{pmatrix},$$

$$A_M = \begin{pmatrix} 0 & 1 & 1 \\ -\infty & 0 & -\infty \\ -\infty & -\infty & 0 \end{pmatrix}.$$

Пустая позиция даёт тождественное преобразование

$$I = \begin{pmatrix} 0 & -\infty & -\infty \\ -\infty & 0 & -\infty \\ -\infty & -\infty & 0 \end{pmatrix}.$$

Если для двух соседних отрезков хранятся матрицы A и B, то для их объединения в дереве отрезков хранится max-plus произведение

$$B \otimes A,$$

то есть сначала применяется левый отрезок, потом правый.

В корне дерева лежит преобразование для всей строки, а ответ получается применением этого преобразования к начальному вектору

$$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

Как поддерживать строку $L/R/M$ онлайн

Это вторая часть полного решения.

Дерево отрезков строится **по порядку номеров**, а не по текущему порядку в ряду. В листе с номером x хранится одна из следующих сущностей:

- I, если участник ещё не добавлен;
- I, если это текущий глобальный минимум;
- матрица `L`;
- матрица `R`;
- матрица `M`.

Иными словами, глобальный минимум просто не кодируется.

Какие символы могут измениться после вставки

Если новый участник добавляется слева, то измениться могут только текущие префиксные минимумы:

- часть символов `L` перестаёт быть `L` и становится `M`;
- если новый участник стал новым глобальным минимумом, старый глобальный минимум превращается из пустой позиции в `R`.

Если новый участник добавляется справа, всё симметрично:

- часть символов `R` превращается в `M`;
- если появился новый глобальный минимум, старый глобальный минимум становится `L`.

Именно это позволяет обновлять строку быстро.

Что хранить дополнительно

Храним две монотонные деки по номерам:

- `pref` — все текущие префиксные минимумы в порядке возрастания номера;
- `suff` — все текущие суффиксные минимумы в порядке возрастания номера.

Первый элемент в обеих деках — текущий глобальный минимум.

Вставка слева

Пусть добавляется участник x.

1. Пока в конце `pref` стоит номер больше x, он перестаёт быть префиксным минимумом:
   - если это не старый глобальный минимум, его тип меняется `L` $\to$ `M`;
   - если это старый глобальный минимум, его тип меняется $I \to$ `R`.
2. После этого возможны два случая:
   - если x — новый глобальный минимум, его лист остаётся равным I, а сам x добавляется в начало обеих дек;
   - иначе x получает тип `L` и добавляется в конец `pref`.

Вставка справа

Полностью симметрично:

- работаем с деком `suff`;
- обычные элементы меняются `R` $\to$ `M`;
- старый минимум при необходимости становится `L`.

Почему это амортизированно быстро

Каждый участник:

- не более одного раза становится префиксным минимумом и не более одного раза перестаёт им быть;
- не более одного раза становится суффиксным минимумом и не более одного раза перестаёт им быть.

Значит, суммарно по всем запросам будет только $O(n+q)$ удалений из дек.

Каждое изменение типа — это одна точечная модификация листа в дереве отрезков, то есть $O(\log(n+q))$.

Итого суммарная сложность:

$$O((n+q)\log(n+q)).$$

Что меняется при $t = 2$

Идея остаётся той же самой:

1. текущий набор по-прежнему кодируется строкой над $\{L, R, M\}$;
2. эта строка по-прежнему поддерживается двумя монотонными деками;
3. дерево отрезков по-прежнему строится по порядку номеров.

Меняется только то, **какую информацию** нужно хранить в вершине дерева.

Почему трёх чисел уже недостаточно

Для $t = 1$ хватало знать:

- лучший уже готовый путь;
- лучший путь, который можно продолжить через левую границу;
- лучший путь, который можно продолжить через правую границу.

Для $t = 2$ этого уже мало. Теперь будущее зависит ещё и от того, как текущая частичная конфигурация взаимодействует с двумя граничными вершинами.

Интуитивно нужно различать такие вещи:

- могут ли левая и правая границы быть концами будущего оптимального пути;
- находятся ли они сейчас в одной компоненте или в разных;
- если в одной, соединены ли они уже частью допустимого пути;
- какую конфигурацию ещё можно корректно продолжить в один простой путь при добавлении новых вершин.

То есть состояние по-прежнему определяется только поведением относительно двух границ, но теперь вариантов этого поведения уже не три, а конечное, но более заметное число.

Автомат

Удобно смотреть на эту ситуацию как на конечный автомат.

Назовём две строки s и t эквивалентными, если для любого продолжения p выполнено

$$f(s + p) - f(s) = f(t + p) - f(t).$$

Иначе говоря, с точки зрения всех будущих продолжений строки s и t ведут себя одинаково, если после них всегда происходят одинаковые переходы и одинаково меняется ответ.

Каждый класс такой эквивалентности можно считать состоянием автомата. При чтении очередного символа из $\{L, R, M\}$ автомат:

- переходит в новое состояние;
- увеличивает текущий ответ на некоторое число.

Для $t = 1$ после минимизации получается 6 состояний. Для $t = 2$ получается 20 достижимых состояний.

Ниже мы будем пользоваться уже готовым автоматом для $t = 2$.

Как получить эти 20 состояний

Выписывать их вручную долго и не нужно. Гораздо удобнее сгенерировать автомат отдельной программой.

Один из стандартных способов такой:

1. перебрать короткие строки над $\{L, R, M\}$;
2. для каждой строки точно посчитать значение $f(s)$ маленьким brute force;
3. считать две строки эквивалентными, если они неразличимы по всем коротким продолжениям;
4. после стабилизации этого разбиения получить конечный набор достижимых состояний и таблицу переходов.

В этой задаче стабилизация происходит быстро, и для случая $t = 2$ получается 20 состояний. После этого таблицу переходов можно просто захардкодить в решение.

Что хранить в вершине дерева отрезков

Для каждого состояния q храним результат применения всего отрезка:

$$go[q] = (\text{newState},\ \Delta),$$

где

- newState — состояние, в которое попадём после чтения всех символов этого отрезка;
- Δ — насколько увеличится ответ.

Для листа с символом `L`, `R` или `M` это просто фиксированная таблица переходов. Для пустой позиции или текущего глобального минимума хранится тождественное преобразование:

$$go[q] = (q, 0).$$

Как объединять две вершины

Пусть для левого сына известно

$$go_L[q] = (mid[q],\ add_L[q]),$$

а для правого сына известно

$$go_R[q] = (to_R[q],\ add_R[q]).$$

Тогда для родителя:

$$mid = go_L[q].state,$$

$$go[q].state = go_R[mid].state,$$

$$go[q].add = go_L[q].add + go_R[mid].add.$$

То есть это просто композиция преобразований: сначала читаем левый отрезок, потом правый. Если q_0 — стартовое состояние автомата, то ответ в корне равен

$$go_{\text{root}}[q_0].add.$$

Инициализация полного решения

Если начальный ряд уже задан, то перед обработкой запросов делаем следующее:

1. строим для него строку $L/R/M$ с нуля;
2. выставляем тип каждой уже присутствующей вершины:
   - минимум $\to I$;
   - остальные $\to L/R/M$;
3. одновременно собираем деки `pref` и `suff`:
   - все `L` и текущий минимум попадают в `pref`;
   - все `R` и текущий минимум попадают в `suff`.

После этого все запросы уже обрабатываются онлайн.

Сложность

Пусть

$$N = n + q.$$

Тогда:

- каждая вершина меняет тип лишь константное число раз;
- суммарно происходит $O(N)$ изменений символов;
- каждое изменение — это точечное обновление дерева отрезков.

Следовательно:

- для $t = 1$ время работы равно
$$O(N \log N),$$
память —
$$O(N);$$
- для $t = 2$ асимптотика та же самая, потому что число состояний автомата постоянно:
$$O(N \log N)$$
по времени и
$$O(N)$$
по памяти.

Итог

Вся задача распадается на две почти независимые части:

1. перевести текущий набор участников в строку над $\{L, R, M\}$;
2. быстро посчитать ответ по этой строке.

Для маленьких подзадач строку можно пересчитывать с нуля. Для полного решения:

- строка поддерживается двумя монотонными деками;
- значение по строке поддерживается деревом отрезков:
  - из max-plus матриц размера 3×3 для $t = 1$;
  - из преобразований автомата из 20 состояний для $t = 2$.

Это и даёт итоговую сложность

$$O((n + q) \log(n + q)).$$

Решение 2

Скажем, что элемент массива имеет тип L, если нет элемента левее него с меньшим значением. Аналогично определим элементы типа R. Заметим, что глобальный минимум удовлетворяет обоим типам, поэтому его в решении будем рассматривать отдельно. Все остальные элементы, которые не удовлетворяют данным типам, будут иметь тип M.

Рассмотрим, как устроены рёбра, исходящие из каждого из элементов в этом графе:

- Если элемент имеет тип L, то из него ведёт ровно одно ребро, ведущее в максимальный по значению элемент типа R, не превосходящий данный.
- Если элемент имеет тип R, то из него ведёт ровно одно ребро, ведущее в максимальный по значению элемент типа L, не превосходящий данный.
- Если элемент имеет тип M, то из него ведёт два ребра в максимальные по значению элементы типов L и R.

Решим сначала версию с ориентированным графом. Будем рассматривать элементы в порядке возрастания значений. Элементы, имеющие тип M, проигнорируем, а другие элементы равных типов объединим в блоки подряд идущих. Минимальный элемент для удобства можно добавить в отдельный блок. Нетрудно видеть, что самый длинный путь из элементов типов L и R определяется однозначно, и равен количеству блоков, идущих раньше (по значениям) чем блок, в который попал соответствующий элемент. Таким образом, длина самого длинного пути, начинающегося в элементах типа L или R имеет длину, равную количеству блоков -1.

Пути, начинающиеся из элементов типа M, сразу же попадают в один из блоков, поэтому чтобы учесть самый длинный такой путь, достаточно рассмотреть элемент типа M с максимальным значением.

Чтобы эффективно поддерживать блоки, можно поддерживать элементы типов L и R в двух стеках рекордов и, при добавлении элемента с одной из сторон, надо удалить суффикс элементов соответствующего стека и добавить в него новый элемент. Таким образом, для решения задачи достаточно уметь обрабатывать следующее:

- Изменить тип элемента с L или R на M.
- Добавить элемент с типом L или R.
- Найти количество блоков, описанных выше.
- Найти количество блоков, идущий до максимального элемента типа M.

Блоки можно поддерживать с помощью std::set. Чтобы избежать других структур данных для последнего запроса достаточно не явно находить количество блоков, а проверять, правда ли, что максимальный элемент типа M (если такой элемент существует) по значению больше минимального элемента в последнем блоке. Данное решение можно реализовать за $\mathcal{O}((n+q)\log(n+q))$.

Данное решение обобщается до случая с неориентированным графом. Будем так же поддерживать блоки, но теперь понадобится дополнительная информация. Назовём стоимостью элемента типа L или R количество элементов типа M больших по значению, чем данный, но меньших, чем ближайший больший элемент типа L или R. Для каждого элемента будем поддерживать его стоимость. Это можно сделать с помощью дерева фенвика или с помощью std::set, если заметить, что все стоимости $\geqslant 3$ в данной задаче эквивалентны, поэтому достаточно рассматривать не более 3х ближайших больших элементов типа M для каждого.

Далее будем относить элементы типа M к блоку, в который попадает элемент типа L или R, в стоимости которого он учитывается. Можно показать, что самый длинный путь начинается в одном из двух последних блоков и заканчивается в одном из двух первых блоков, не считая минимума, или в самом минимальном элементе. Чтобы учесть все случаи, достаточно для каждого блока хранить стоимость первого/последнего и дополнительно двух максимальных по стоимости элементов в нём. Далее есть два пути:

- Рассмотреть много случаев и обновить через них ответ. Описание всех случаев заняло бы очень много места, поэтому они в разборе опущены.
- Заметить, что мы можем оставить $\mathcal{O}(1)$ элементов из первых и последних двух блоков, а так же минимальный элемент. От всех остальных блоков важна только суммарная стоимость максимальных элементов в них. Таким образом, задачу можно свести к троичной строке не очень

большой длины. Для всех таких строк можно заранее посчитать ответ перебором или динамикой. Такой способ позволяет избежать ручной обработки всех случаев, что сильно упрощает реализацию.

Такое решение можно реализовать за $\mathcal{O}((n+q)\log(n+q))$ с большой константой.