

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 $i \rightarrow p_i$, 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 $a_i \geq 0$, 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 ($n \leq 1000$)

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 $b[0 \dots m - 1]$ в порядке следования по ребрам $i \rightarrow p_i$. После отключения t ($0 \leq t \leq k$) 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 $len = 1 \dots \min(m, k + 1)$ и считаем сумму $b[l] + b[l + 1] + \dots + b[l + len - 1]$ (индексы берём по модулю m). Сохраняем глобальный максимум.

Суммарная сложность $O(\sum (|C|^2)) \leq O(n^2)$ 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 $a_i \geq 0$, 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 каждом 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: $n \leq 150\,000$; $k = 1$

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

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

Воспользуемся методом динамического программирования: вычислим значения $dp[v][s]$ — 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 $dp[v][s]$ 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 $\sum_v \prod_{u_1 \neq u_2} sz(u_1)sz(u_2)$, где суммирование ведется по всем вершинам v , а u_1 и u_2 — непосредственные потомки вершины v , и $sz(w)$ 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: $n \leq 150\,000$, $w_i = 1$

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 искомым ответ на задачу.

Подзадача 7: $n \leq 150\,000$, 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:

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

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. $n \leq 500\,000$

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 $s_1, \dots, s_n$ — 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 $s_1, \dots, s_n$ в том же порядке. Ограничения — длина w должна быть $O(\sum |s_i|)$, конкретные ограничения на длину указаны в подгруппах.

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

В этих подгруппах можно закодировать данные строки $s_1, \dots, s_n$ как « $(s_1 \dots s_n)$ ». 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 $s'_1, \dots, s'_n$ из полученной ПСП « $(s'_1 \dots s'_n)$ ».

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

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

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 $|s_1| \leq |s_2|$, 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: $f(x) = 4 \cdot x + 2000$, $S \leq 200\,000$, $n = 2$

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:

$$((s_1)((s_2)(\dots((s_n)((()))))\dots)))$$

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 $s \geq 2n$, $m = s + 4n + 6 \leq 3s + 6$, 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: $f(x) = x + 2000$, $t_1 = t_2 = \dots = t_n > 1$

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 поддерева, 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 (87811) 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 87811, 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

$$x_1 < x_2 < \dots < x_k$$

в порядке возрастания, а их позиции в ряду равны

$$p_1, p_2, \dots, p_k.$$

Тогда:

- x_1 — текущий глобальный минимум, его пропускаем;
- дальше поддерживаем

$$mn = p_1, \quad mx = p_1;$$

- для каждого $i \geq 2$:
 - 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: $n + q \leq 16$

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:

$$dp[mask][v] = \text{существует ли путь, использующий } mask \text{ и заканчивающийся в } v.$$

Сложность одного пересчёта равна

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

где m — текущее число участников.

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

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

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 $n, q \leq 1000$.

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:

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

Символ R. Симметрично:

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

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

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

Для пустой строки:

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

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:

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

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 перестает 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 $L \rightarrow M$;
 - если это старый глобальный минимум, его тип меняется $I \rightarrow R$.
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 $R \rightarrow M$;
- старый минимум при необходимости становится L .

Почему это амортизировано 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

$$go_L[q] = (\text{mid}[q], \text{add}_L[q]),$$

а для правого сына известно

$$go_R[q] = (\text{to}_R[q], \text{add}_R[q]).$$

Тогда для родителя:

$$\text{mid} = go_L[q].\text{state},$$

$$go[q].\text{state} = go_R[\text{mid}].\text{state},$$

$$go[q].\text{add} = go_L[q].\text{add} + go_R[\text{mid}].\text{add}.$$

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

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

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 $\rightarrow I$;
 - остальные $\rightarrow L/R/M$;
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 ≥ 3 в данной задаче эквивалентны, поэтому достаточно рассматривать не более $3x$ 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 динамической. 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.

Задача 5. Обобщённые шахматы

Краткое условие задачи Дана доска $n \times n$ ($1 \leq n \leq 400$), клетка (i, j) покрашена в цвет a_{ij} . Для каждой пары r, c ($1 \leq r, c \leq n$) нужно найти минимальное количество клеток для перекраски b_{rc} , необходимое, чтобы прямоугольник из первых r строк и первых c столбцов стал *правильно раскрашенным*:

- используется не более двух разных цветов;
- соседние по стороне клетки имеют разные цвета.

Краткая идея решения

Для фиксированного прямоугольника (r, c) нужно выбрать два цвета c_0, c_1 (для клеток с чётной и нечётной суммой координат). Для каждой чётности нам нужны два самых частых цвета. В полном решении используется сжатие цветов и поддержка двух максимумов за $\mathcal{O}(1)$, что даёт $\mathcal{O}(n^3)$ времени на полное решение.

Подробный разбор:

Подзадача 1. ($n \leq 50$)

Вместо того, чтобы считать количество клеток, которые нужно перекрасить — будем наоборот считать количество клеток, которые не поменяют цвет. Количество таких клеток мы хотим максимизировать. У нас есть два типа клеток (i, j) — у которых $(i+j) \bmod 2 = 0$ и $(i+j) \bmod 2 = 1$, и они должны быть разных цветов. Зафиксируем подпрямоугольник (r, c) , для которого мы решаем задачу. Нам нужно зафиксировать два цвета col_1 и col_2 , чтобы максимизировать количество клеток с $(i+j) \bmod 2 = 0$ цвета col_1 + количество клеток с $(i+j) \bmod 2 = 1$ цвета col_2 .

Сделаем структуру данных, в которую мы будем уметь добавлять пару (col, cnt) и поддерживать два максимальных cnt у разных цветов col . Переберем все клетки подпрямоугольника (r, c) и посчитаем для каждого цвета и типа клетки (остатка по модулю два) количество вхождений. После этого посчитаем два максимума для клеток типа 1 и типа 2 (с помощью структуры, которую мы определили выше). Теперь, если самый частый цвет среди клеток типа 1 и клеток типа 2 различаются, то нам выгодно оставить их. Иначе нам нужно рассмотреть два варианта: второй по частоте цвет среди клеток типа 1 + самый частый цвет среди клеток типа 2, и самый частый цвет среди клеток типа 1 + второй по частоте цвет клеток типа 2.

Возьмем максимум среди этих возможных вариантов, и ответом для подпрямоугольника (r, c) будет rc — количество клеток, которые сохранили цвет. Асимптотика $\mathcal{O}(n^4 \log n)$, где $\log$ для того, чтобы считать для каждого цвета количество его вхождений.

Подзадача 2. ($n \leq 200$)

Эта подзадача близка к полному решению, но в ней можно реализовать структуру, которая поддерживает два максимума за $\log$, либо реализовать структуры данных не самым оптимальным образом.

Зафиксируем r и найдем ответ для всех подпрямоугольников (r, c) . Будем перебирать c по возрастанию и поддерживать для каждого цвета и типа клетки, количество вхождений этого цвета в текущий подпрямоугольник (r, c) . Также, будем для обоих типов поддерживать два самых частых цвета, как в подзадаче 1. При переходе от c к $c+1$ честно добавим r клеток, которые добавляются при расширении прямоугольника.

Таким образом, получаем решение за $\mathcal{O}(n^3 \log n)$, где $\log$ от допустимой неэффективной реализации структуры с двумя максимума на базе `map` или `set`.

Подзадача 3. (Только два различных цвета)

В этой подзадаче всего два цвета, поэтому для каждого подпрямоугольника у нас всего две возможные цветовые конфигурации и мы можем явно посчитать ответ для каждой из них за $O(n^2)$.

Подзадача 4 (не более 10 разных цветов)

В этой подзадаче мы можем перебрать пару цветов, в которые будет раскрашена доска и для каждого подпрямоугольника проверить конфигурацию за $O(n^2)$. Таким образом, время работы $O(10^2 \times n^2)$.

Подзадача 5 (не более 100 разных цветов)

Эту группу тестов могли проходить решения с недостаточно эффективной реализацией следующих подгрупп.

Подзадача 6 (не более 10000 разных цветов)

В этой подзадаче ожидается полное решение, за исключением того, что чтобы поддерживать количество вхождений какого-то цвета, достаточно создать массив на 10000 элементов, вместо поддержки счетчиков за логарифм.

Полное решение

Сначала сожмем координаты, то есть сделаем так, чтобы все числа оказались меньше, чем n^2 . Сжатие координат можно реализовать с помощью хеш-таблицы или с помощью сортировки и бинарного поиска. Этой займет время $O(n^2)$ или $O(n^2 \log n)$. После этого реализуем решение из подзадачи 2, в котором мы за $O(1)$ поддерживаем количество вхождений какого-то цвета и за $O(1)$ поддерживаем два максимума. Получаем решение за $O(n^3)$, которое без проблем сдаётся.

Задача 6. Ночь, улица, фонарь, аптека

Подгруппа 1: $t = 1$, $n \leq 10$

Переберём все подмножества фонарей. Осталось проверить, что подмножество фонарей является экономным. Для этого можно перебрать для каждого фонаря, смотрит он вправо или влево, и явно проверить, выполняется ли условие их экономности. Данный перебор работает за $O(3^n)$, так как каждому перебранному варианту может соответствовать троичная маска: 0 - мы не взяли фонарь в экономное множество, 1 — мы взяли фонарь, и он направлен влево, 2 - мы взяли фонарь, и он направлен вправо.

Подгруппа 2: $t = 1$, $x_i - s_i \neq x_j$ и $x_i + s_i \neq x_j - s_j$

Описанное условие означает, что все фонари на отрезке нужно направить вправо. Будем считать $dp_{i,x}$ — количеством подмножеств фонарей, если мы рассматриваем фонари с номерами $\geq i$, и подмножество можно направить вправо так, чтобы самый левый освещенный отрезок начинался в позиции x .

При пересчёте $dp_{i,x} = dp_{i+1,x}$, кроме ситуации, когда мы используем i -й фонарь. В таком случае получается обновление $dp_{i,x_i} = 1 + dp_{i+1,x_i+s_i}$.

Получаем решение за $O(n + X)$, где X — максимальное ограничение на координаты.

Подгруппа 3: $t = 1$, $s_i \neq s_j$

Рассмотрим, как устроен произвольный ответ. Сначала сколько-то первых фонарей у нас направлены вправо. Далее, если дальше фонарь был направлен влево, то теперь следующие фонари должны освещать его правый конец, тогда все оставшиеся фонари светят влево. Таким образом, у нас сначала сколько-то фонарей направлены вправо, а потом сколько-то влево.

Заметим, что в этой подгруппе каждое подмножество можно определить единственным способом (кроме случая, если оно состоит ровно из одного фонаря). Если у нас было направление и вправо, и влево, то ориентация однозначна, а иначе у нас все фонари направлены в одну сторону, и, чтобы можно было развернуть, надо, чтобы все s_i были одинаковыми.

Тогда посчитаем ответ следующим образом: посчитаем dp аналогично второй подгруппе, это найдёт количество способов начать с левой границы в x . Посчитаем аналогичное dp в обратном порядке по правым границам, тогда мы найдём количество способов закончить в x .

Теперь переберём точку, где соприкасаются префикс и суффикс, и перемножим значения dp . Не забудем прибавить все отдельные значения dp , чтобы учесть направления только в одну сторону.

Осталось вычесть из ответа n , так как мы два раза учли подмножество из одного фонаря.

Получаем решение за $O(n + X)$, где X — максимальное ограничение на координаты.

Подгруппа 4: $t = 1$, $s_i = s_j$

Аналогично предыдущей подгруппе посчитаем две dp . Теперь поймём, как устроен ответ: это либо все направлены в одну сторону, тогда надо просуммировать значения любой из dp , либо у нас есть фонари направленные и вправо, и влево, тогда надо перебрать точку соприкосновения и перемножить.

Получаем решение за такую же асимптотику.

Подгруппа 5: $t = 1$, $n \leq 1000$

Напишем $dp_{i,x}$ — количество способов выбрать подмножество экономных фонарей вместе с их ориентацией, если мы рассмотрели только первые i фонарей, а последняя освещённая координата x . Тогда переходы в этой динамике следующие:

- $dp_{i+1,x_i} = dp_{i,x_i-s_i} + 1$. Мы направляем фонарь i влево.
- $dp_{i+1,x_i+s_i} = dp_{i,x_i} + 1$. Мы направляем фонарь i вправо.

Осталось понять, какие состояния мы учли несколько раз. Экономное множество фонарей может учитываться несколько раз, если каждый из фонарей можно ориентировать в другую сторону, и подмножество останется экономным. Очевидно, что в таком множестве все фонари будут направлены в одну сторону, поскольку иначе будет разрыв. Тогда $i_1, i_2, \dots, i_k$ можно ориентировать как влево, так и вправо, если выполняется $x_{i_1} + s_{i_1} = x_{i_2}, x_{i_2} + s_{i_2} = x_{i_3}, \dots, x_{i_{k-1}} + s_{i_{k-1}} = x_{i_k}$ и $s_{i_1} = s_{i_2} = \dots = s_{i_k}$.

Тогда ответ на задачу будет суммой всех значений dp минус количество таких подмножеств. Динамика считается за $O(n^2)$ или за $O(n^2 \cdot \log(n))$, если динамику хранить в `std::map`. А количество таких множеств тоже считается за $O(n^2)$, так как максимальная такая «цепочка» имеет длину n , и мы можем её явно перебрать.

Подгруппа 6: $t = 1$

Осталось научиться считать за линейное время dp , а также количество «цепочек». Для пересчёта динамики можно заметить, что для очередного i меняются только 2 состояния по x . Поэтому можно хранить dp_x — количество экономных подмножеств, оканчивающихся в точке x . И для очередного i пересчитать следующим образом все состояния:

- $dp_{x_i+s_i} := dp_{x_i+s_i} + dp_{x_i} + 1$.
- $dp_{x_i} := dp_{x_i} + dp_{x_i-s_i} + 1$.

Для подсчёта цепочек можно написать dp_i^2 — количество «цепочек», которые кончаются на i -м фонаре. Тогда для того, чтобы пересчитать количество «цепочек», оканчивающихся на j -м фонаре, достаточно проверить, есть ли фонарь в координате $x_j - s_j$, и, если он есть (пусть его номер i), тогда, если $s_i = s_j$, то $dp_j^2 = dp_i^2 + 1$, иначе $dp_j^2 = 1$.

Подгруппа 7: $t = 2$, если $x_i = x_{i+1}$, то $s_i \neq s_{i+1}$

Также посчитаем динамику из подгруппы 6. Тогда если у нас в очередной координате ровно один фонарь, то переходы остаются такими же, иначе они следующие. Пусть номер фонаря, для которого мы пересчитываем состояния, равен i , первый из его прожекторов имеет силу освещения s_1 , а второй s_2 . Тогда:

- $dp_{x_i+s_1} := dp_{x_i+s_1} + dp_{x_i} + dp_{x_i-s_2} + 2$. Первый прожектор ориентировали вправо.
- $dp_{x_i+s_2} := dp_{x_i+s_2} + dp_{x_i} + dp_{x_i-s_1} + 2$. Второй прожектор ориентировали вправо.
- $dp_{x_i} := dp_{x_i} + dp_{x_i-s_1} + dp_{x_i-s_2} + 2$. Ровно один из прожекторов ориентировали влево.

Осталось также посчитать, сколько есть подмножеств, у которых можно поменять ориентацию, чтобы подмножество осталось экономным. Но при двух прожекторах на одном фонаре у нас может быть более сложная конструкция: мы можем поменять между собой направления двух прожекторов местами. Поэтому для учёта «цепочек» мы можем написать $dpchain_{x,y}$ — количество «цепочек», которые заканчиваются в координате x , и, если мы поменяем ориентацию прожекторов (если один прожектор светил влево, другой вправо, то после смены ориентации будет наоборот) у последнего фонаря, то последняя освещённая координата увеличится на y . Тогда в ней переходы следующие в случае (мы говорим, что $s_1 > s_2$):

- $dpchain_{x_i,s_1} := dpchain_{x_i,s_1} + 1 + dpchain_{x_i-s_1,s_1}$. Мы в фонаре поставили прожектор s_1 , который светит влево.
- $dpchain_{x_i,s_2} := dpchain_{x_i,s_2} + 1 + dpchain_{x_i-s_2,s_2}$. Мы в фонаре поставили прожектор s_1 , который светит вправо.
- $dpchain_{x_i+s_2,s_1-s_2} := dpchain_{x_i+s_2,s_1-s_2} + 1 + dpchain_{x_i-s_1,s_1-s_2}$. Мы в фонаре поставили прожектор s_2 , который светит вправо, и прожектор s_1 , который светит влево. Тогда, если мы поменяем ориентацию двух фонарей, то станет покрыта координата $x_i + s_1$.

В случае, если у фонаря ровно один прожектор, то переходы будут следующие:

- $dpchain_{x_i,s_i} := dpchain_{x_i,s_i} + 1 + dpchain_{x_i-s_i,s_i}$.

Эту динамику можно хранить в `std::map<int, map<int, int>`. Тогда заметим, что все «цепочки» были посчитаны в этой динамике, так как, если мы меняем местами прожекторы (для лучшего понимания можно считать, что, если был поставлен один прожектор, то было поставлено два прожектора, один из которых освещает отрезок длины 0), то каждый отрезок, освещённый одним фонарём, сдвигается на дельту сил прожекторов. А значит, ответ — сумма всех состояний dp минус сумма всех состояний $dpchain$.

Полное решение:

В последней подгруппе требуется понять, что нужно немного поменять переходы:

- $dp_{x_i+s_i} := dp_{x_i+s_i} + 2 \cdot dp_{x_i} + dp_{x_i} + 3$.
- $dp_{x_i} := dp_{x_i-s_i} + 2 \cdot dp_{x_i} + dp_{x_i} + 2$.

А $dpchain$ пересчитывается следующим образом:

$$dpchain_{x_i,s_i} := dpchain_{x_i,s_i} + 2 + 2 \cdot dpchain_{x_i-s_i,s_i}.$$

Задача 7. Марсианский рюкзак

Переформулировка задачи

Для каждого префикса из первых i предметов нужно найти минимально возможное значение побитового OR выбранного подмножества, у которого суммарная стоимость не меньше C . Если такого подмножества нет, ответ равен -1 .

Подзадача 1: $n \leq 20$, $k \leq 10$

Можно просто перебрать все подмножества первых i предметов для каждого i . Для каждого подмножества считаем сумму стоимостей и побитовое OR странностей. Среди подмножеств с суммой не меньше C берём минимальный OR.

Асимптотика $\mathcal{O}(2^0 + 2^1 + \dots + 2^n) = \mathcal{O}(2^{n+1})$.

Подзадачи 2 – 3: $n \leq 50\,000$, $k \leq 10$

Для фиксированной маски x максимальная суммарная стоимость подмножества с общей странностью $y \subseteq x$ равна $F_i(x) = \sum_{j \leq i, w_j \subseteq x} c_j$, где запись $w_j \subseteq x$ означает, что w_j является подмаской x , то есть $(w_j \& \sim x) = 0$.

Значит, ответ для префикса i — это минимальная маска x , для которой $F_i(x) \geq C$.

Будем поддерживать значения $F_i(x)$ напрямую: когда приходит предмет с маской w_i и стоимостью c_i , нужно прибавить c_i ко всем маскам x , для которых $w_i \subseteq x$. После этого достаточно найти минимальную маску x с $F_i(x) \geq C$.

Асимптотика $\mathcal{O}(n \cdot 2^k)$.

Подзадачи 4 – 5: $n \leq 1\,000\,000$, $k \leq 19$, w_i — степени двойки или $n \leq 2000$

В данных подзадачах, чтобы найти нужный ответ для префикса p , можно идти жадно по битам сверху вниз. Пусть вначале в маске разрешены все биты, а текущая доступная суммарная стоимость равна $T = \sum_{i=1}^p c_i$. Далее идём по битам от старших к младшим и пытаемся удалить текущий бит, за $\mathcal{O}(p)$ находим T' — новую сумму c_i по подходящим предметам. Если $T' < C$, то возвращаем бит, иначе оставляем его нулем. Далее переходим к меньшему биту.

Асимптотика $\mathcal{O}(n^2 k)$, что достаточно для решения подзадачи 4.

Для решения подзадачи 5 можно просимулировать алгоритм выше, заметив, что T' можно вычислить как вычитание из T суммы c_i с w_i , равными нужной степени двойки.

Подзадача 6: $n \leq 500\,000$, $k \leq 16$

Для данной подзадачи нужно применить технику «Meet In The Middle».

Каждую маску w_i представим в виде $w_i = (h_i, \ell_i)$, где:

- h_i — старшие p бит,
- ℓ_i — младшие q бит.

Аналогично ответ будем искать в виде $ans_i = (H_i, L_i)$, где сначала минимизируем старшую часть H_i , а затем при фиксированной H_i минимизируем младшую часть L_i .

Для нахождения старшей части применяем решение из подзадачи 3 за $\mathcal{O}(n \cdot 2^p)$. Тогда при добавлении нового предмета H_i либо не изменяется (остается равным H_{i-1}), либо уменьшается.

Если $H_i < H_{i-1}$, то пересчитаем значение функции $F_i(x)$ для всех $H_i \cdot 2^q \leq x < (H_i + 1) \cdot 2^q$. Для этого по всем j , таким, что $1 \leq j \leq i$, и если $h_j \subseteq H_i$, то в $F_i(H_i \cdot 2^q + \ell_j)$ добавим c_j , а далее сделаем SOS-DP для F в промежутке $[H_i \cdot 2^q, (H_i + 1) \cdot 2^q)$.

Иначе, если $H_i = H_{i-1}$ и $h_i \subseteq H_i$, то обновим нужные значения F за $\mathcal{O}(2^q)$.

Итого, так как уменьшений H не более $\mathcal{O}(2^p)$, то решение суммарно работает за $\mathcal{O}(2^k \cdot q + n \cdot (2^p + 2^q))$. При $p = q = 2^{\frac{k}{2}}$ решение работает за $\mathcal{O}(2^k \cdot k + n \cdot 2^{\frac{k}{2}})$.

Полное решение

Будем строить ответы сразу для целого отрезка префиксов и сразу по всем битам. Пусть старшие биты ответа уже зафиксированы, а осталось определить ещё t младших бит.

Для текущего рекурсивного вызова будем хранить:

- число **pref** — уже зафиксированная часть ответа в старших битах;
- отрезок префиксов $[L, R]$, для которых все ответы имеют один и тот же уже построенный старший префикс;
- массив cnt длины 2^t .

Массив cnt построен по первым $L - 1$ предметам. Для каждой маски s длины t значение $cnt[s]$ равно суммарной стоимости тех предметов, которые:

- среди уже обработанных старших битов не противоречат маске **pref**, то есть их старшие биты являются подмаской соответствующих битов ответа;
- на оставшихся t младших битах точно равны s .

Иначе говоря, мы сгруппировали все предметы, которые ещё могут входить в ответ, по их оставшейся младшей части.

Как определить очередной бит? Пусть сейчас мы определяем старший из оставшихся t бит, то есть бит с номером $t - 1$ внутри текущего массива cnt .

Разобьём все маски на две половины:

- нижняя половина — маски, у которых этот бит равен 0;
- верхняя половина — маски, у которых этот бит равен 1.

Если мы хотим поставить текущий бит ответа в 0, то брать можно только предметы из нижней половины. Максимальная суммарная стоимость таких предметов равна $S_0 = \sum_{s < 2^{t-1}} cnt[s]$.

Отсюда сразу следует критерий:

- если $S_0 \geq C$, то существует допустимое решение с текущим битом 0;
- если $S_0 < C$, то никакое решение с текущим битом 0 невозможно, значит, этот бит обязан быть равен 1.

Этого достаточно, так как, если текущий бит ответа равен 0, то остальные более младшие биты мы в лучшем случае можем поставить в 1. Значит, мы действительно учитываем все предметы, которые можно взять при текущем бите 0.

Где разбивается отрезок префиксов? Из-за монотонности ответов существует такая граница M , что

- для префиксов $L, L + 1, \dots, M - 1$ текущий бит ответа равен 1;
- для префиксов $M, M + 1, \dots, R$ текущий бит ответа равен 0.

Найдём её одним проходом.

Изначально массив cnt уже соответствует первым $L - 1$ предметам, а значение $S_0 = \sum_{s < 2^{t-1}} cnt[s]$ соответствует тем из них, которые можно брать при текущем бите 0.

Теперь будем последовательно рассматривать предметы с номерами $L, L + 1, \dots, R$. Для очередного предмета i проверим, может ли он участвовать в решении текущего узла:

- на уже зафиксированных старших битах его маска не должна содержать единиц там, где в `pref` стоит ноль;
- текущий рассматриваемый бит у него должен быть равен нулю, если мы хотим считать сумму S_0 .

Как только после добавления очередного предмета можно впервые получить $S_0 \geq C$, мы нашли позицию M .

Время поиска границы на одном узле равно $\mathcal{O}(R - L + 1)$.

После того как граница M найдена, остаются две рекурсии.

Левая часть рекурсии отвечает за префиксы $[L, M - 1]$. Теперь текущий бит уже зафиксирован как единица, значит, на этом бите предмет может иметь и 0, и 1. Следовательно, для каждой маски младших $t - 1$ бит нужно просто сложить две половины: $next_1[s] = cnt[s] + cnt[s + 2^{t-1}]$ для $0 \leq s < 2^{t-1}$.

Стоимость построения массива для левой части: $\mathcal{O}(2^t)$.

Правая часть рекурсии отвечает за префиксы $[M, R]$. В этом случае брать можно только нижнюю половину: $next_0[s] = cnt[s]$ для $0 \leq s < 2^{t-1}$.

Но есть важный нюанс: массив для правого сына должен соответствовать уже первым $M - 1$ предметам, а массив cnt был построен только для первых $L - 1$.

Поэтому во время поиска границы M мы одновременно поддерживаем и массив $next_0$: все подходящие предметы с текущим битом 0, встретившиеся на позициях от L до $M - 1$, добавляются в нужную ячейку по своим младшим $t - 1$ битам.

Тогда к моменту запуска правой рекурсии массив уже готов.

Остаётся отдельно обработать префиксы, для которых ответа вообще не существует. Это делается заранее: найдём первый префикс, у которого суммарная стоимость всех предметов стала хотя бы C . Для всех предыдущих префиксов выводим -1 , а рекурсию запускаем только с этого места.

Итого, решение за $\mathcal{O}(nk + 2^k k)$.

Задача 8. Хорошие раскраски — 8

Подзадача 1.

В случае $n = 3$ ответ равен $k(k - 1)$, если все рёбра активны и k в ином случае.

Подзадача 2.

Пусть Γ_i ($1 \leq i < m$) — это грань, содержащая на своей границе ребро $l_i \rightarrow l_{i+1}$, а Γ_0 — это грань, содержащая на своей границе ребро $l_m \rightarrow l_1$.

Тогда можно сначала раскрасить грань Γ_0 , на это есть k способов, затем останется $(k - 1)$ свободных цветов для грани Γ_1 , а если продолжить процесс раскраски граней в порядке $\Gamma_2, \dots, \Gamma_m$ далее, то для каждой очередной грани будет доступно $(k - 2)$ цветов.

Таким образом, в зависимости от n ответ на задачу будет равен:

$$\begin{aligned} \text{answer} &= k, & n &= 3 \\ \text{answer} &= k(k - 1)(k - 2)^{\frac{n-5}{2}}, & n &\geq 5 \end{aligned}$$

Подзадача 3.

В этом случае, если все рёбра не активны, то ответ равен k , а в ином случае, если будет активно ровно x рёбер, то грани объединятся в x граней, которые соединены по циклу друг с другом. Количество способов раскрасить x таких граней в k цветов можно вычислить с помощью метода динамического программирования:

- $\text{dp}_1[i]$ — количество способов раскрасить i граней в k цветов так, чтобы любые две грани i и $i + 1$ имели различные цвета, а также чтобы первая и последняя грань имели различные цвета.
- $\text{dp}_2[i]$ — количество способов раскрасить i граней в k цветов так, чтобы любые две грани i и $i + 1$ имели различные цвета, но при этом первая и последняя грани должны иметь один и тот же цвет.

Очевиден базовый случай $\text{dp}_1[1] = 0$, $\text{dp}_2[2] = k$. А переходы расписываются так:

$$\begin{aligned} \text{dp}_1[i] &= \text{dp}_1[i - 1] \cdot (k - 2) + \text{dp}_2[i - 1] \cdot (k - 1) \\ \text{dp}_2[i] &= \text{dp}_1[i - 1] \end{aligned}$$

Подзадачи 4–5.

В эти подзадачи заходили переборные решения. Например, авторское решение имеет время работы $\mathcal{O}((q + 1) \cdot (n^3 + k^n \cdot n))$ и устроено следующим образом:

- Явно строится граф соседних граней в исходном графе.
- С помощью алгоритма Флойда-Уоршелла грани разбиваются на группы, которые должны объединиться в одну грань.
- Перебираются всевозможные k^n раскрасок исходных граней и явно проверяются на корректность.

Подзадача 6.

Для решения этой подзадачи достаточно было заметить, что ответ равен 0, если в дереве есть вершина нечётной степени, и равен 2 во всех остальных случаях.

Подзадача 7.

Замечание. Здесь и далее мы будем обозначать $\chi(n)$ — как количество способов раскрасить n граней, соединённых по циклу, используя k цветов.

Совместим идеи из 2-й и 3-й подзадач: сначала раскрасим грань Γ_0 , которая *накрывает* дерево сверху, а затем будем раскрашивать грани в порядке их вложенности. Причём грани, ограниченные сверху одной и той же вершиной дерева, будем раскрашивать одновременно.

Если вершина v является корнем дерева, то требуется раскрасить $\deg v - 1$ граней, это можно сделать $\frac{\chi(\deg v)}{k}$ способами. Если же v не является корнем, то под ней находится $\deg v - 2$ граней, и их можно раскрасить $\frac{\chi(\deg v)}{k(k-1)}$ способами. Таким образом, требуется перемножить эти величины по всем нелистовым вершинам v , полученное значение и будет ответом на задачу.

Подзадачи 8–10.

Назовём вершину v *висячей*, если она не является листом и соединена не более чем одним активным ребром с другими вершинами. Заметим, что от удаления *висячей* вершины ответ не изменяется, поэтому идея решения этих подзадач заключается в том, чтобы отвечать на запросы независимо и наиболее эффективным образом уметь избавляться от таких вершин.

Авторское решение предлагает воспользоваться тем, что в задаче явно задан массив предков $p_i < i$, а поэтому избавиться от *висячих* вершин можно в два прохода:

- Сначала перебираем вершины в порядке от n до 1, если вершина v является листом, то она не висячая и она не удаляется. В ином случае вершина удаляется только тогда, когда в неё не ведут активные рёбра из неудалённых вершин с большими номерами.
- Во втором проходе будем рассматривать вершины v в порядке от 1 до n . Если вершина v является корнем своей компоненты (то есть из неё наверх не исходит активное ребро в неудалённую вершину) и при этом эта вершина имеет единственного сына, то такая вершина удаляется.

После двух таких проходов в дереве не остаётся висячих вершин. Но, возможно, дерево распадается на компоненты связности. Заметим, что мы всё также можем раскрашивать грани в порядке вложенности, и каждая вершина v , которая является корнем, будет давать множитель $\frac{\deg' v}{k}$ к ответу, а все остальные нелистовые вершины будут давать множитель $\frac{\deg' v}{k(k-1)}$.

Представленное решение имеет асимптотику $O(n \cdot q)$, а также быстро работает, так как отсутствуют прыжки по памяти и вся работа, по сути, совершается в двух-трёх циклах `for`.

Подзадача 11.

В этой подзадаче предлагается сжать исходное дерево до размера $4q$ так, чтобы ответ от этого не изменился с точностью до домножения на константный множитель M . Делать это достаточно сложно, так как требуется проверять достаточно много случаев, поэтому при самостоятельной реализации этой идеи могут потребоваться стресс-тесты.

Общая идея заключается в том, чтобы разбить все рёбра на два класса: *постоянные* и *нестабильные*. Постоянными мы назовём рёбра, которые активны на протяжении всех запросов, нестабильными те рёбра, которые изменяют своё состояние в одном из запросов.

Заведём массив e , где $e(v)$ — количество вершин на внешнем цикле, которые соединены постоянными рёбрами с v . Для инициализации положим, что $e(v) = 1$ для листьев v и $e(v) = 0$ для всех остальных вершин. Соответственно, мы перестаём считать, что цикл проходит именно через листья дерева, а предполагаем, что он проходит через какие-то другие (назовём их *виртуальными*) вершины, которые учтены в значениях $e(v)$. Ясно, что от этого ответ на задачу не изменится.

Далее вычислим для каждой вершины v величину:

- $\min f(v)$ — максимальное количество непересекающихся путей, которые ведут вниз по дереву от вершины v в виртуальные вершины по постоянным рёбрам.

Тогда можно сжать дерево, используя следующие манипуляции:

- Выделить цепочки вершин вида $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_l$, где каждая пара соседних вершин в цепочке соединена ребром и вершины $v_2, \dots, v_{l-1}$ имеют ровно двух соседей. Тогда такие цепочки можно объединить в одно ребро, соответствующим образом модифицировав последовательность запросов — новое ребро будет активным тогда и только тогда, когда активны все рёбра из исходной цепочки.

Затем следует выделить *опорные* вершины, то есть те вершины v , для которых выполнено $\min f(v) \geq 2$. Все вершины v , из которых по рёбрам, ведущим вверх, достижима какая-то отличная от v опорная вершина и для которых выполнено $\min f(v) \geq 1$, будем называть *предсказуемыми*. Такие вершины интересны тем, что они никогда не будут удалены в ходе последовательного удаления висячих вершин.

Поэтому каждую предсказуемую вершину можно переподвесить к виртуальному корню -1 , а в её бывшем предке p увеличить значение $e(p)$ на единицу, это никак не изменит ответ на задачу. Вершину -1 для удобства мы будем считать опорной (но в ответе для неё множитель считать не будем, так как она не является реальной вершиной, а выступает удобной записью для факта, что нам не важно, куда именно подвешена вершина v). Если вершина v подвешена за вершину -1 и $\min f(v)$ совпадает с количеством её сыновей, то такая вершина будет давать постоянный вклад в ответ, который можно учесть в множителе M , после чего, удалив эту вершину из дерева, всех её сыновей переподвесить к вершине -1 .

Подробный анализ показывает, что после сжатия дерева таким образом останется не более $4q$ вершин. Таким образом, решить эту подзадачу можно за время $O(n + q^2)$.

Подзадача 12.

Попробуем оптимизировать решение за $O(nq)$ другим способом, воспользовавшись тем, что высота дерева h не превосходит 20. Идея решения заключается в том, чтобы явно поддерживать дерево, которое остаётся после удаления висячих вершин (см. решение подзадач 8–10). Для этого достаточно разделить вершины на три типа:

- N — вершины, удалённые после первого прохода;
- S — вершины, удалённые после второго прохода;
- P — вершины, которые не будут удалены в ходе алгоритма.

Теперь рассмотрим процесс удаления ребра $v \rightarrow p_v$. Сначала следует пробежаться по активным рёбрам вверх по вершинам, начав с вершины p_v , до того момента, пока в очередную вершину не будет входить ещё какое-то активное ребро снизу, соединяющее её с вершиной P -класса, либо пока мы не дойдём до корня дерева. Все посещённые вершины перейдут в класс N .

Если мы остановились в вершине, в которую ведёт ещё одно ребро, то в случае, если это ребро единственное, следует дальше продолжать подниматься вверх по активным рёбрам. Если мы остановились из-за того, что из очередной вершины u не исходит активного ребра вверх, то требуется спуститься вниз от вершины u до первой вершины, которая будет листом или у которой будет два сына P , соединённых с ней активными рёбрами. Все посещённые вершины следует отметить классом S . Если же мы дошли до вершины, в которую ведёт ещё одно активное ребро из вершины типа P , то не требуется дополнительно изменять классы вершин.

Так как мы изменяем классы у вершин по одной, то можно с лёгкостью пересчитывать ответ.

Данный алгоритм работает за время $O(n + q \cdot h)$.

Подзадачи 13–15.

Есть два подхода для решения задачи на полный балл, оба работают за время $O(n + q \log n)$, но основаны на разных принципах.

Подход 1. (D&Q) Заметим, что можно слегка модифицировать решение подзадачи 11 (требуется дополнительно обрезать из каждой компоненты стабильное или нестабильное ребро, ведущее вверх, которое гарантированно состоит из висячих вершин) и применить его в методе разделяй и властвуй.

Сперва сожмём дерево, а затем разделим запросы на две примерно равные половины и решим задачу для этих половинок запросов рекурсивно. То есть будем сжимать дерево под запросы и снова делить их на две равные группы. Так как дерево, соответствующее отрезку из q запросов, по размеру не превышает $O(q)$, то такое решение будет работать за $O(q \log q)$.

Подход 2. (Структуры данных): Основан на том, чтобы поддерживать сжатое дерево на вершинах, оставшихся в P