Задача A. Бесплатная и платная операции

Переберём i от 0 до n и посчитаем, сколько монет нам придётся потратить, если в первой операции мы выберем $x = i$. Для $x = 0$ это просто количество единиц в массиве a. Для $x = i$ с $i \geqslant 1$, если $a_i = 1$, то нам нужно будет потратить на одну монету меньше, чем при $x = i - 1$, потому что при $x = i - 1$ мы тратили одну монету на то, чтобы заменить i-й элемент массива на 1, а при $x = i$ её тратить не нужно. Если же $a_i = 0$, то нам придётся потратить на одну монету больше, потому что при $x = i - 1$ мы, наоборот, не тратили монету на то, чтобы сделать i-й элемент массива равным 0, а при $x = i$ её нужно будет потратить. Посчитаем таким образом минимальное количество потраченных монет для всех x и выберем из них наименьшее.

Асимптотика: $\mathcal{O}(n)$.

Задача B. Круглая парковка и пачки машин

Считаем все описания желаемых парковочных мест и запомним для каждого места, сколько машин хотят его занять. После этого распределим машины по местам. Пройдёмся по парковочным местам по возрастанию их номеров. Если на текущее место претендует больше одной машины, то оставим на нём только одну, а остальные передвинем на следующее место. Пройдёмся так по всем местам от нулевого до $n - 1$-го два раза. При этом любая машина, вне зависимости от того, с какого места она начинала движение, сможет проехать мимо всех парковочных мест, а значит, где-то припаркуется. После того как все машины были распределены по местам, пройдёмся по местам ещё раз и найдём номер минимального свободного.

Асимптотика: $\mathcal{O}(n)$.

Задача C. Привет от Снарка

Обозначим за s_i строку из ввода, а за $digit_i$ — тот десятичный разряд, к которому относится цифра s_i. Например, цифры I и V будут относиться к десятичному разряду единиц (первому разряду), X и L — к разряду десятков (второму разряду), и так далее.

Заметим, что задача чем-то похожа на поиск лексикографически максимальной НВП: нам нужно найти такую последовательность римских цифр, что первые несколько из них относятся к какому-то i-му десятичому разряду и составляют корректную десятичную цифру этого разряда, следующие несколько — к другому десятичному разряду j, младше, чем i, и так далее. Причём нам нужно, чтобы цифр в числе было максимальное количество, и из всех таких последовательностей нас интересуют те, у которых i максимально, а из них те, у которых максимальна десятичная цифра в i-м разряде, а потом j, цифра разряда j, и так далее.

Будем искать самое длинное число так же, как мы бы искали НВП наивным алгоритмом за $\mathcal{O}(n^2)$. Будем идти по символам строки в обратном порядке, чтобы потом мы могли найти лексикографически максимальный ответ. Насчитаем $dp_{i,value}$ — такую наибольшую длину, что с суффикса i можно взять римское число этой длины, которое начинается с цифры s_i, и эта цифра вместе с другими цифрами, относящимися к тому же десятичному разряду, образуют десятичную цифру $value$. Например, если $s_i = X$, то в $dp_{i,1}$ сохраним длину самого длинного римского числа, заканчивающегося на X, в $dp_{i,2}$ — на XX, и так далее для XXX, XL и XC. Также запомним ту $dp_{j,value'}$, из которой мы обновили нашу $dp_{i,value}$, и из всех таких dp с одинаковой длиной выберем ту, у которой пара $(digit_j, value')$ лексикографически больше.

Чтобы корректно обработать девятки, заведём фейковый разряд 0, и в $dp_{i,0}$ для цифр $s_i = X, C, M \ldots$ будем хранить такую наибольшую длину, что есть римское число этой длины, в котором перед s_i нет ни одной цифры, относящейся к десятичному разряду $digit_i$ или $digit_{i-1}$. Тогда мы сможем пересчитать $dp_{j,9}$ через $dp_{i,0}$. Например, если $s_i = M$, то в $dp_{i,0}$ мы будем запоминать такие числа, которые заканчиваются на M и у которых нет других цифр, относящихся к разряду тысяч или сотен, и если s_j будет равно C, то мы сможем пересчитать $dp_{j,9}$ через $dp_{i,0}$ и получить число, оканчивающееся на CM — десятичную цифру 9 разряда сотен.

Наивная реализация будет работать за $\mathcal{O}(n^2)$. Чтобы её ускорить, будем, подобно тому, как мы поддерживаем ДО для поиска НВП, поддерживать массив $opt\_dp_{digit,value}$, в котором будем хранить оптимальную $dp_{i,value}$ среди всех таких i, что $digit = digit_i$. Если оптимальных dp несколько, будем сохранять ту, у которой пара $(digit_j, value')$ лексикографически наибольшая, а при их равенстве

будем запоминать ту dp, которую мы позднее всех обновляли, так как в ней потенциально учтены числа, лексикографически большие в поздних разрядах. Теперь пересчёт одной dp_i работает за $digit_i \cdot 10 \sim 260$, так как нам нужно будет перебрать все $opt\_dp_{digit_j,value'}$ с $digit_j < digit_i$, чтобы обновить $dp_{i,0}$, $dp_{i,1}$ и $dp_{i,5}$, и ещё какое-то константное количество $opt\_dp_{digit_i \pm 1,value'}$ для обновления остальных $dp_{i,value}$. Чтобы ускорить пересчёт до перебора $26 + C$ состояний, сохраним для каждого $digit$ оптимальное $opt\_dp_{digit,value}$ среди всех возможных $value$ в ещё одном вспомогательном массиве.

Теперь пересчёт работает за $\mathcal{O}(n \cdot C)$ с константой C порядка 30.

Задача D. Без AC

Очевидно, что когда мы удаляем из строки два подряд идущих символа, один из них стоит на чётной позиции, а другой — на нечётной. Более того, при удалении двух подряд идущих символов чётность позиций всех остальных не меняется: каждый символ либо остаётся на той же позиции, либо сдвигается на две позиции назад.

Преобразуем строку. Каждую букву A, стоящую на нечётной позиции, заменим на C, а каждую C, стоящую на нечётной позиции, — на A. В силу вышесказанного, правило при этом преобразуется таким образом: можно удалить любые два соседних символа, кроме AA или CC. Преобразовать в пустую по такому правилу можно те и только те строки, в которых символов A и C не больше, чем по $\frac{n}{2}$.

Всего строк из трёх символов 3^n. Вычтем из них те строки, в которых символов A или C больше $\frac{n}{2}$. Их количество можно вычислить как $\Sigma_{i=\frac{n}{2}+1}^{n} 2 \cdot C_n^i \cdot 2^{n-i}$: переберём, каких символов будет больше половины: A или C; сколько именно символов A или C будет в строке, рассмотрим все способы расставить их на n мест, а на остальные места поставим любой из двух оставшихся символов.

Чтобы посчитать эту сумму за $\mathcal{O}(n)$, предпосчитаем факториалы и обратные факториалы всех чисел до n, а также все степени двойки до n-й (или же будем аккуратно поддерживать текущую степень двойки при суммировании).

Задача E. Игра гномов

Поставим себя на место врага и посчитаем, сколько урона мы получим от гномов. На первом ходу нас атакуют все гномы, на втором — все, кроме тех, которых мы убьём на первом ходу, на третьем — кроме тех, которых мы убьём на первых двух ходах, и так далее. Тогда нам выгодно на каждом своём ходу метать молнию в ту группу, в которой больше всего гномов, так как в этом случае мы сильнее всего снизим суммарный урон по нам.

Исходя из этого, все стоящие перед нами группы гномов можно представить как несколько «целых» блоков гномов по k в каждом, которых мы убьём одной молнией, и несколько «остатков» размера $< k$, на каждый из которых нам также придётся потратить по молнии. В частности, группа из x гномов даёт $\lfloor \frac{x}{k} \rfloor$ целых блоков и остаток размера $x \,\%\, k$. Нашей стратегией будет начать с целых блоков, а затем идти по остаткам в порядке убывания их размера.

Следовательно, задачу можно эквивалентно переформулировать так: сколько максимально урона можно нанести, если разделить гномов на несколько групп размера ровно k и на m, возможно, пустых, групп размера $< k$.

Количество групп из k гномов варьируется от $\lceil \frac{n-m(k-1)}{k} \rceil$ до $\lfloor \frac{n}{k} \rfloor$. Среди них не более чем $\lceil \frac{m(k-1)}{k} \rceil \leqslant m$ различных количеств. Переберём, сколько их будет, и распределим остальных гномов по группам размера $< k$ как можно оптимальнее.

Утверждается, что распределять гномов нам нужно примерно поровну, то есть так, чтобы их количество в группах отличалось не более, чем на один. Действительно, пусть в какой-то группе x гномов, и это хотя бы на два гнома больше, чем в другой, где гномов y. Для удобства отсортируем группы по убыванию количеств гномов в них. Мы можем считать, что враг метает молнии в группы именно в таком порядке. Перенесём одного гнома из самой правой группы по x в самую левую группу по y. При этом группы гномов останутся отсортированы по возрастанию количества, но тот гном, которого мы перенесли, проживёт дольше, а значит, нанесёт больше урона.

Разделить n гномов на такие группы можно только одним образом — на $n \,\%\, m$ групп по $\lfloor \frac{n}{m} \rfloor + 1$ гномов и на $n - n \,\%\, m$ групп по $\lfloor \frac{n}{m} \rfloor$ гномов. Используя формулы суммы арифметической прогрессии,

можно посчитать, сколько урона нанесут гномы при таком распределении.

Асимптотика: $\mathcal{O}(m)$.

Задача F. Разложить строку на две одинаковые

Применим технику meet-in-the-middle. Насчитаем все такие подпоследовательности u, что первую половину строки можно разделить на подпоследовательности v и vu. Для этого переберём маску тех индексов, которые мы выберем, насчитаем подпоследовательности из выбранных и невыбранных букв и проверим, что одна является префиксом другой. Аналогично насчитаем все такие подпоследовательности u, что вторая половина строки делится на w и uw. Если среди этих наборов подпоследовательностей есть общая u, то строку можно разбить на две подпоследовательности vuw, и ответ Yes, а иначе ответ No.

Осталось только научиться быстро проверять, есть ли такая u. Для этого можно посортировать строки в обоих наборах и пройтись по отсортированным спискам двумя указателями, как при сортировке слиянием, и проверять на равенство пары строк, на которых останавливаются указатели. Чтобы добиться лучшей асимптотики, можно захешировать строки и сортировать хеши.

Асимптотика: $\mathcal{O}(2^{\frac{|s|}{2}} \cdot |s|)$.

Задача G. Экотропы, но уже без легенды

Для каждого ребра графа важны два параметра (x, c) — то, соединяет ли он важную вершину и неважную, и его вес. Нам нужно найти такое остовное дерево, что сумма первых параметров равна w, а сумма вторых как можно меньше.

Чтобы это сделать, применим лямбда-оптимизацию. Пусть вес ребра (x, c) будет равен $x \cdot \lambda + c$, где λ — какая-то константа. Запустим алгоритм Краскала поиска минимального остовного дерева на рёбрах с такими весами и посчитаем, сколько ребёр с первым параметром, равным 1, он выберет. Чем больше λ, тем меньше ребёр с первым параметром 1 выберет алгоритм, и наоборот. В частности, при $\lambda = -\infty$ Краскал максимизирует количество выбранных рёбер с первым параметром 1, а при $\lambda = +\infty$ оно будет минимально возможным. Если w больше максимально возможного значения или меньше минимально возможного, то ответ — -1. Иначе подберём бин. поиском такое λ, что алгоритм выберет ровно w ребёр с первым параметром, равным 1.

Покажем, что функция $f(x) =$ «вес мин. остова графа с рёбрами веса $a_i \cdot x + c_i$, $a_i = 1$ или 0» является выпуклой вверх, из чего будет следовать, что такое λ возможно подобрать. Действительно, $f(x)$ — это минимум из всех прямых $A \cdot x + C$, таких, что A и C — это сумма коэффициентов a_i и c_i у ребёр, образующих мин. остов. Каждая такая прямая — выпуклая вверх функция, а минимум выпуклых вверх функций также выпуклый вверх. Значит, лямбда-оптимизация в этом случае работает.

Асимптотика — $\mathcal{O}(m \log m \log C)$, если при каждом запуске Краскала пересортировывать рёбра по новым весам. Однако можно добиться и $\mathcal{O}(m \log C)$, если заранее отдельно отсортировать по возрастанию веса рёбра с первым параметром, равным 1, и с первым параметром, равным 0, и перед запуском Краскала мёрджить два списка.

Задача H. ПСП в прямоугольники

Чтобы посчитать ответ, возьмём все «внешние» чёрные прямоугольники, то есть те, которые касаются внешней неограниченной белой области, просуммируем их площади, исключим из суммы площади белых прямоугольников, находящихся внутри тех чёрных, из площади белых исключим площади всех чёрных прямоугольников, находящихся внутри тех белых, и так далее. В итоге ответ будет равен сумме площадей всех чёрных прямоугольников минус сумма площадей всех белых.

Чтобы его найти, для каждой пары соответствующих друг другу скобок определим ширину, высоту и цвет их прямоугольника. Чтобы их определить, пройдём по символам ПСП со стеком. Каждую открывающую скобку будем класть на стек, а когда нам приходит закрывающая скобка, будем брать со стека верхнюю открывающую скобку, смотреть на их прямоугольник и соответствующим образом учитывать его в ответе. Определить ширину прямоугольника легко — это разность индексов закрывающей и открывающей скобки. Чтобы её узнать, достаточно положить на стек

вместе с открывающей скобкой её индекс. Цвет можно определить в тот момент, когда мы кладём на стек открывающую скобку. Если в этот момент стек пуст, то цвет будет чёрным, а иначе он будет противоположен цвету открывающей скобки с верхушки стека. Чтобы определить высоту прямоугольника, будем для каждой открывающей скобки, лежащей на стеке, самую высокую точку геометрического представления, которое мы нарисовали после этой скобки. Когда мы кладём скобку на стек, проинициализируем эту высоту нулём. После того как скобка снимается со стека, мы рисуем новый прямоугольник на 1 выше самой высокой точки, а значит, нам нужно обновить эту высоту для скобки, которая сейчас находится на вершине стека.

Асимптотика: $\mathcal{O}(n)$.

Задача I. Кратчайшая программа для робота

Рассмотрим произвольную программу для робота. Выпишем номера строк с командами в том порядке, в котором робот будет их исполнять. Возможны два варианта: либо робот исполнит несколько различных строк $l_1, l_2, \ldots, l_n$, после чего перейдёт к команде $l_k, k \leqslant n$, которую уже исполнял, либо же робот исполнит каждую команду не более одного раза. В первом случае робот выполнит команды $l_1, l_2, \ldots, l_{k-1}$ по одному разу, а потом будет вечно исполнять команды $l_k, l_{k+1}, \ldots, l_n$ по циклу. Во втором случае робот не зациклится, и программа рано или поздно завершится.

То есть, какой бы ни была программа, её можно переписать либо вообще без команд $G(i)$ (в том случае, если она останавливается), либо с ровно одной такой командой в самом конце программы (если она начинает выполнять набор команд по циклу), и при этом переписывании количество команд в программе не увеличится. Поэтому имеет смысл рассматривать только программы таких двух видов.

Чтобы найти оптимальную останавливающуюся программу, запустим поиск в ширину. Минимальное расстояние от старта до финиша и будет минимальной длиной останавливающейся программы.

Чтобы найти оптимальную программу с циклом, переберём ту клетку (i_0, j_0), на которой программа впервые дойдёт до строки l_k и войдёт в цикл. После этого будем перебирать, на сколько клеток по вертикали и горизонтали (D_i, D_j) мы сдвинемся за одну итерацию цикла, и сколько полных итераций цикла K мы сделаем до того момента, как дойдём от (i_0, j_0) до финиша. Зная эту информацию, мы можем посчитать минимальное количество команд в нецикличной части программы (это минимальное расстояние от стартовой клетки до (i_0, j_0), которое ищется поиском в ширину) и минимальную длину цикличной части программы. Чтобы найти её, нам нужно запустить поиск в ширину от клетки (i_0, j_0), но при этом не заходить в клетку (i, j), если хотя бы одна из клеток (i, j), $(D_i + i, D_j + j)$, …, $((K-1)D_i + i, (K-1)D_j + j)$ сломана. Также, цикличная часть программы должна доходить до финишной клетки на $K+1$-й итерации. То есть, мы также не имеем права заходить в клетку (i, j), если клетка $(KD_i + i, KD_j + j)$ сломана и при этом мы ни разу не зашли в клетку $(i_F - KD_i, j_F - KD_j)$, где (i_F, j_F) — финишная клетка. Чтобы посчитать длину цикличной части, достаточно запустить ещё один поиск в ширину по описанным правилам, в котором, помимо текущей клетки, хранить то, посетили ли мы клетку $(i_F - KD_i, j_F - KD_j)$.

Избавимся от запуска `BFS`-а из каждой клетки (i_0, j_0). Будем по-прежнему перебирать (D_i, D_j) и K. Разделим цикл на две части: на ту, которая повторится $K+1$ раз, и на ту, которая повторится K раз. Чтобы для каждой клетки (i_0, j_0) найти минимальную длину той части, которая выполняется $K+1$ раз, запустим `BFS` из финишной клетки, при этом при прохождении через клетку (i, j) будем проверять, не сломаны ли клетки $(i - kD_i, j - kD_j)$ для k от 0 до K. Кратчайшая длина этой части пути для клетки (i_0, j_0) — это расстояние от клетки (i_F, j_F) до $(i_0 + KD_i, j_0 + KD_j)$. Для той части пути, которая выполняется K раз, будем запускать `BFS` из клетки $(i_F - D_i, j_F - D_j)$ и проверять те же клетки для k от 0 до $K-1$. Для клетки (i_0, j_0) кратчайшая длина этой части пути — расстояние от $(i_F - D_i, j_F - D_j)$ до $(i_0 + KD_i, j_0 + KD_j)$.

При наивной реализации каждый `BFS` будет работать за $\mathcal{O}(NMK)$, так как в каждой клетке необходимо сделать K проверок. Но его можно ускорить до $\mathcal{O}(NM)$. Для этого заранее предпосчитаем для всех (i, j), верно ли, что хотя бы одна из клеток (i, j), $(i + D_i, i + D_i)$, …, $(i + (K-1)D_i, j + (K-1)D_j)$ сломана. Чтобы быстро это пересчитать, в тот момент, когда мы в переборе увеличиваем K на один, пройдёмся по всем NM клеткам и пометим клетку (i, j) как

недоступную для посещения, если клетка $(i + (K-1)D_i, j + (K-1)D_j)$ сломана для нового K.

Докажем, что интересующих нас троек $(D_i, D_j), K$ на самом деле не $\mathcal{O}(NM(N+M))$, а $\mathcal{O}(NM)$. Действительно, для $K = 1$ нас интересуют все D_i, не превосходящие по модулю N, и все D_j, не превосходящие по модулю M. Но для $K = 2$ нас уже не интересуют D_i с $|D_i| > \frac{N}{2}$ и D_j с $|D_j| > \frac{M}{2}$, так как клетка $(i + KD_i, j + KD_j)$ гарантированно будет недоступной для посещения. Аналогично, для произвольного K нас интересуют лишь порядка $\frac{NM}{K^2}$ значений (D_i, D_j). Сумма $\sum_{K=1}^{max(N,M)} \frac{NM}{K^2} = \mathcal{O}(NM)$.

Асимптотика: $\mathcal{O}(NM)$ запусков `BFS`-а и $\mathcal{O}(NM)$ на `BFS`, всего: $\mathcal{O}(N^2M^2)$.

Задача J. Пары карточек

Как известно, максимальное количество пар, которое можно сформировать в таком массиве — это $\min(S - \max_{i=1}^{n} a_i, \frac{S}{2})$, где $S = \sum_{i=1}^{n} a_i$. Причём если $\max_{i=1}^{n} a_i \geqslant \frac{S}{2}$, то их можно сформировать лишь одним способом — найти такой номинал i_{max}, что $a_{i_{max}}$ максимально среди всех a_i, и образовать по a_j пар (i_{max}, j) для всех j.

Идея решения такая: переберём все пары (i, j) в лексикографическом порядке. Для каждой пары будем добавлять в ответ такое максимальное количество этих пар, что $\max_{i=1}^{n} a_i$ не становится больше или равным $\frac{S}{2}$. Для нахождения этого количества можно использовать бинарный поиск. В тот момент, когда после взятия пары в ответ $\max_{i=1}^{n} a_i$ стало равно $\frac{S}{2}$, остановимся и образуем пары единственным образом.

Заметим, что если мы будем перебирать лишь такие пары (i, j), где a_i и a_j ненулевые, то после их обработки произойдёт одно из трёх событий: карточки i закончатся, карточки j закончатся или же $\max_{i=1}^{n} a_i$ станет равно $\frac{S}{2}$. Так как события 1 и 2 могут суммарно за всё время работы программы произойти лишь n раз, а событие 3 — один, то мы переберём лишь $\mathcal{O}(n)$ пар.

Чтобы реализовать решение, нам потребуется структура, которая будет хранить все a_i, а также будет уметь выкидывать произвольное число карточек одного номинала и проверять, не стало ли $\max_{i=1}^{n} a_i$ больше или равно $\frac{S}{2}$. Для этого она должна поддерживать S и уметь быстро находить максимум среди a_i. Также структура должна уметь искать лексикографически минимальную пару (i, j), для которой $a_i, a_j \neq 0$. Чтобы поддерживать все операции, достаточно завести *map*, в котором по i будет храниться a_i, при этом ключи, у которых $a_i = 0$, из него удалять, и *multiset* с текущим набором a_i.

Асимптотика: $\mathcal{O}(n \log n \log C)$.

Задача K. Ожерелье

Применим `XOR`-хеширование. Для каждого вида бусин выберем какую-нибудь бусину этого вида. Всем бусинам, за исключением выбранных, назначим хеш — случайное 64-битное число. После этого каждой бусине из выбранных назначим хеш, равный `XOR`-у хешей остальных бусин этого вида. Теперь `XOR` хешей всех бусин одного вида равен нулю. По условию, вырезать бусины $i, i+1, \ldots i+k$ из ожерелья можно тогда и только тогда, когда для каждого вида бусин, который присутствует в ожерелье, среди $i, \ldots, i+k$ встречаются либо все бусины этого вида, либо ни одной. Заметим, что если это условие выполнено, то `XOR` хешей этих бусин равен нулю, а если нет, то с высокой вероятностью он не равен нулю. Насчитаем p_i — `XOR` хешей бусин на позициях от 1 до i. Тогда сделать разрезы после i-й и j-й бусин можно в том случае, когда $p_i = p_j$.

Чтобы найти первую часть ответа, посчитаем для каждого x, сколько есть i таких, что $p_i = x$, и просуммируем C_{cnt}^2 по всем этим количествам, предварительно сжав координаты в массиве p_i. Чтобы найти вторую часть ответа, выпишем для каждого x все индексы i с $p_i = x$ и пройдёмся по ним двумя указателями так, чтобы расстояние между индексами, на которых стоят указатели, было как можно ближе к $\frac{n}{2}$, и запомним самое близкое расстояние.

Асимптотика: $\mathcal{O}(n \log n)$.

Задача L. Прямые и вложенные круги

Так как каждый следующий круг целиком покрывает предыдущий, то прямая, которая имеет общую точку с i-м кругом, имеет её и со всеми кругами, идущими после i-го. Найдём для каждой прямой при помощи бин. поиска первый круг, с которым она имеет общую точку. Для этого

посчитаем расстояние от центра рассматриваемого в бин. поиске круга до прямой и сравним его с радиусом круга.

Для i-го круга запомним a_i — для скольких прямых он является первым кругом с общей точкой. Ответ на задачу — это префиксные суммы массива a.

Асимптотика: $\mathcal{O}(q + n \log q)$.

Задача M. База знаний Холмса

Назовём *возможным вариантом развития событий* такое множество событий, что все они могли одновременно произойти, и вместе с этим остальные события могли не произойти. Формально, это значит, что:

- В множестве есть все события s_i.
- Если в множестве есть событие A, и в базе знаний Холмса есть вывод $A \to B$, то B там тоже есть.
- Если в множестве есть событие B, и в базе знаний Холмса есть хотя бы один вывод $A \to B$, то хотя бы одно из таких A присутствует в множестве.

Научимся за $\mathcal{O}(m)$ проверять для события v, точно ли оно произошло или нет. Для этого попробуем найти такой вариант развития событий, что v в нём не произошло. Найдём все такие события, которые также не могут произойти в этом варианте. Для этого нужно найти все такие события u, что из u по цепочке выводов следует v: $u \to a_1 \to a_2 \to \ldots \to v$. Чтобы их найти, построим граф, вершинами в котором будут состояния, а рёбрами — обратные выводы (для вывода $A \to B$ добавим в граф ребро $B \to A$). Запустим в этом графе обход в глубину от вершины v. Очевидно, что из всех событий, достижимых в этом обходе, следует v по цепочке выводов, и только из них. Пометим их как не произошедшие в искомом варианте.

Если среди них оказалось событие s_i, то v обязательно произошло. Иначе, попробуем найти для каждого события s_i его причину (то есть, событие, удовлетворяющее третьему пункту). Для этого запустим обход в глубину из каждой вершины s_i, не заходя в события, которые мы пометили как не произошедшие. Если мы смогли дойти до события, которому не нужна причина (то есть, до такого B, что выводов $A \to B$ нет), то это событие и события, лежащие на пути от него до s_i, могли произойти, так как они удовлетворяют пункту 3. Чтобы найти ответ для всех вершин из s_i, насчитаем на вершинах графа, не помеченных как не произошедшие, динамику: dp_w = «достижима ли из w такая вершина, из которой не выходит никаких ребёр». Насчитать её можно обходом в глубину. Если $dp_{s_i} = 1$ для всех s_i, то искомый вариант развития событий найден, и v могло не произойти. Иначе его не существует, и v точно произошло.

Асимптотика: $\mathcal{O}(dm)$.

Задача N. Массив с двойками

Заметим, что если какой-то индекс i упоминается хотя бы в одном наблюдении с $c \neq x$, то $a_i \neq 0$. Пометим все такие a_i, которые не могут быть равны 0. Для остальных элементов массива можно смело ставить $a_i = 0$, потому что a_i либо не упоминается ни в одном наблюдении, и нам безразлично, чему он равен, либо фигурирует в наблюдениях i j `x`, и тогда мы ничего не испортим, если сделаем его равным 0.

Чтобы найти остальные значения a_i, применим ту же технику, что применяется при решении `2-SAT`. Построим граф, в котором вершинами будут $i_{\frac{1}{2}}$, i_1 и i_2, означающие, что $a_i = \frac{1}{2}, 1$ или 2 соответственно. Проведём в нём рёбра-следствия из наблюдений. Например, для наблюдения i j $+$, в котором $i \neq j$, оба a_i и a_j должны быть равны 1 или 2, при этом хотя бы один из них должен быть равным 2. Чтобы учесть это в графе, проведём рёбра $i_{\frac{1}{2}} \to i_1$ и $j_{\frac{1}{2}} \to j_1$, что будет означать, что из $a_i = \frac{1}{2}$ следует противоречие $a_i = 1$, а ещё проведём рёбра $i_1 \to j_2$, $j_1 \to i_2$, которые означают, что при $a_i = 1$ должно быть $a_j = 2$, и наоборот. Аналогичные действия сделаем и для всех остальных наблюдений.

Запустим в полученном графе алгоритм выделения компонент сильной связности и их топологической сортировки. Для каждого i посмотрим на ту вершину i_x, которая находится в самой

поздней в топологической сортировке компоненте сильной связности. Если вместе с ней находится ещё какая-то вершина i_y, то из $a_i = y$ следует $a_i = x$, а из $a_i = x$ следует $a_i = y$. Тогда, из-за этих противоречий, $a_i \neq x, y$. Значит, a_i равно тому третьему значению, которое не равно ни x, ни y. Если же в компоненте с i_x нет других вершин i_y, то можно назначить $a_i = x$. Из-за того, что i_x находится в самой поздней компоненте, другие i_y из него не следуют, значит, это не вызывает противоречий с другими возможными значениями a_i.

Заметим, что так же, как и в классическом `2-SAT`, если в нашем графе есть ребро $i_x \to j_y$, то есть и ребро $i_y \to j_x$. Значит, аналогично доказательству корректности `2-SAT`, такое назначение не вызовет противоречий между значениями a_i и a_j при различных i и j.

Асимптотика: $\mathcal{O}(n + m)$.