Задача A. Проверка состояния

Автор задачи: Антон Вдовин, разработчик: Даниил Голов

В **первой подгруппе** все числа в массиве не превосходят 10; $10! = 3\,628\,800$, а значит, подойдёт решение просто посчитать факториал каждого a_i, взять их наибольший общий делитель и проверить на равенство с факториалом $\texttt{gcd}(a)$, который, очевидно, тоже не будет превышать 10!.

Во **второй и третьей подгруппе** имелось всего два или три числа, называем их a, b (и c). Заметим, что $\texttt{gcd}(a!, b!) = \min(a, b)!$. Тогда во второй подзадаче требуется проверить, что $\texttt{gcd}(a, b)! = \min(a, b)!$, что равносильно $\texttt{gcd}(a, b) = \min(a, b)$. Такое выполняется только когда большее из двух чисел делится на меньшее. Для трех чисел решение аналогично.

В **четвертой подгруппе** все числа простые, что означает, что чаще всего их `gcd` равно 1. Единственное исключение — когда все числа одинаковы. Тогда в первом случае достаточно проверить, что `gcd` всех факториалов равен 1, что бывает только тогда, когда $\min(a) = 1$. Во втором же случае все числа массива равны и ответ тогда всегда `YES`.

Далее заметим ключевое для решения задачи наблюдение. Поскольку $x!$ является делителем $(x+1)!$ для любого натурального x, наибольший общий делитель факториалов нескольких чисел — это просто факториал меньшего из них. Действительно, каждый факториал делится на любой меньший факториал, поэтому $\texttt{gcd}(\min(a))$ является их общим делителем. А так как `gcd` не может превосходить никакой из своих аргументов, найденный общий делитель — минимальный.

Для **полного решения** оставалось только полноценно воспользоваться равенством $\texttt{gcd}(a_1!, a_2!, \dots a_n!) = \min(a_1, a_2, \dots a_n)!$. Получаем, что необходимо проверить, что $\min(a)! = \texttt{gcd}(a)!$. Что равносильно тому, что $min(a) = gcd(a)$. Проверка этого условия тривиальна — надо проверить, что каждое число массива делится на минимум массива, что можно сделать за $\mathcal{O}(n)$.

Задача B. Выбор версий компонентов

Автор задачи: Даниил Орешников, разработчик: Павел Скобелин

В **первой подгруппе** имеется ровно одна зависимость. Можно заметить, что в таком случае всем компонентам, которые не зависят от других, можно выбрать одинаковую версию — x, а единственной компоненте, зависящей от какой-то другой — версию $y \geqslant a \cdot x + b$. Так как мы хотим минимизировать сумму версий компонентов, выберем минимально возможную версию $y = a \cdot x + b$. Теперь решим неравенство $\sum_{i=1}^{n} d_i \leqslant X$, значит $x \cdot (n - 1 + a) + b \leqslant X$, и узнаем x.

Во **второй подгруппе** граф зависимостей имел всего 3 компоненты и $X \leqslant 100$, значит можно было перебрать распределение X по трем компонентам, проверив выполнение всех зависимостей.

Далее в **третьей**, **четвертой** и **восьмой подгруппе** было ограничение $X \leqslant 10 \cdot n$. Это довольно сильно ограничивает ответ, ведь несложно показать, что ответ на задачу (то есть, минимальная из версий компонент) будет не более $\frac{X}{n}$, что в этой серии подзадач не превосходит 10. Значит, в этих подзадачах можно было **перебрать** минимальную из версий компонент и проверить, возможно ли расставить версии так, чтобы минимальная версия была такой.

Для подхода к полному решению нужно было воспользоваться бинарным поиском по ответу. Действительно, если существует распределение версий, которое удовлетворяет заданным ограничениям и имеет минимальную версию m, то есть и распределение версий, в котором минимальная версия равна $m - 1$. Аналогично, если не существует ответа с минимумом M, то не существует и ответа с минимумом $M + 1$.

Исходя из соображений выше, мы свели задачу от нахождения минимума к проверке: теперь нужно проверить, есть ли такое распределение версий, в котором минимальная версия равна m. Будем делать это с помощью динамического программирования.

В **третьей и пятой подгруппе** граф зависимостей имел форму бамбука. Мы знаем, что минимальная версия равна m. Тогда поставим ее в лист нашего бамбука и начнем подниматься вверх (к корню), выбирая текущую версию как минимально возможную (то есть, для зависимости u_i, v_i, a_i, b_i, которая образует ребро $u_i \to v_i$, выберем $d_{u_i} = a_i \cdot d_{v_i} + b_i$. В этой и следующих подзадачах необходимо было обрабатывать переполнение — например, если в какой-то момент мы назначили вершине версию $> X$, то нужно было закончить проверку (в таком случае, ответа не существует).

В **четвертой и шестой подгруппе** граф зависимостей имел форму дерева. В таком случае можно было воспользоваться обходом в глубину. Таким образом, для вершины v нужно было рекурсивно запустить обход для всех ее соседей и взять как версию текущей компоненты значение $\max_{u \to v_i}(a_i \cdot v_i + b_i)$, где v_i — это ребра, исходящие из u.

Для решения **седьмой**, **восьмой** и **девятой подгруппы** нужно было воспользоваться ДП на ацикличных неориентированных графах (ДП на DAG'е). Так как граф ацикличный, у него существует топологическая сортировка — давайте рассмотрим ее. Теперь пойдем по ней справа налево и будем считать ДП: сначала поставим в вершину p_i значение m — текущий минимум, который мы проверяем. Тогда значение ДП в текущей вершине это максимум из m и значения линейных функций по исходящим ребрам. Все ребра из этой вершины идут направо, значит все значения ДП, от которых зависит текущее значение, уже посчитаны.

В зависимости от реализации алгоритма это решение проходило 7, 8, 9 подгруппы. Для решения **десятой подгруппы** нужно было аккуратно обработать переполнения, ведь в бинпоиске значение m могло быть порядка X, то есть 10^{18}. Самый простой вариант — тип `int128` в C++. Также для проверки, что произведение чисел a и b больше 10^{18} ($a \cdot b > 10^{18}$) можно использовать другую технику: $a \cdot b > 10^{18} \iff a > \frac{10^{18}}{b}$, что можно проверить в целых числах типа `long long` без переполнения.

Задача C. Полиция 2099

Автор задачи и разработчик: Даниил Орешников

Для начала переформулируем и упростим задачу. Дано дерево, в каждой вершине написана случайная буква. Для строк-запросов необходимо ответить, сколько раз эта строка встречается на каком-то вертикальном пути в дереве.

Как и всегда, **первая подгруппа** расчитана на любой перебор ответа, поэтому мы не будем на ней фокусироваться.

Вторая подгруппа гарантировала $p_i = i - 1$, иными словами, что дерево является бамбуком. На бамбуке задача превращается в стандартную задачу поиска подстроки в строке, что можно решить за линейное время с помощью хешей или алгоритма КМП.

В **третьей подгруппе** был только один шаблон для поиска, и он состоял только из букв '`a`'. Для решения достаточно было с помощью ДП посчитать для каждой вершины число букв '`a`' на пути в нее сверху. Тогда ответом на запрос будет число вершин, у которых полученное значение получилось не меньше $|s_1|$. Действительно, любой вертикальный путь однозначно задается своей нижней вершиной, поэтому подходят те их них, в которые сверху ведет достаточное число букв '`a`'.

Одно из решений **четвертой подгруппы** — полиномиальные хеши. Если для каждой вершины посчитать хеш строки, полученной как путь от корня до нее, то дальше такими хешами можно пользоваться как префиксными хешами в стандартной задаче поиска подстроки в строке, что дает решение за $\mathcal{O}(n + |s_1|)$.

С другой стороны, в **пятой подгруппе** работал самый простой перебор: если все шаблоны короткие, можно проверить вхождение каждого шаблона в каждое возможное место в дереве за линейное время. Аналогично простым решением можно было обойтись и в **шестой подгруппе**: для каждого шаблона и каждого возможного места его вхождения за линейное время проверить их, перебирая символы шаблона по одному.

В **седьмой подгруппе**, исходя из ограничений, проходило решение за $\mathcal{O}(nm)$. Если простой перебор из предыдущей подгруппы не проходил по времени, можно было соединить переборное решение с решением хешами, тогда наличие шаблона с концом в каждой возможной вершине можно было проверять за $\mathcal{O}(1)$. Однако условие о случайности распределения букв по дереву давало неплохой шанс пройти эту подгруппу и простым перебором, если применить какие-то стратегии отсечения или случайного перебора букв вместо последовательного.

Наконец, **полное решение** требовало напрямую воспользоваться фактом случайного распределения букв по дереву. Рассмотрим $K = 5$ и заметим, что вероятность встретить в дереве конкретную строку длины K равна $\frac{1}{26^K}$, что заметно меньше максимального числа вершин. Разобьем все шаблоны на шаблоны длины $< K$ и $\geqslant K$.

- Для первых (коротких) просто сделаем предподсчет, аналогичный пятой подгруппе. Для каждой строки длины от 1 до K посчитаем количество ее вхождений в дерево за $\mathcal{O}(nK)$. Тогда отвечать на такие запросы можно за $\mathcal{O}(1)$.

- Для остальных шаблонов разобьем их на группы по последним K буквам. Матожидание количества вхождений каждого возможного суффикса длины K в дерево меньше 1. Поэтому если при предподсчете разбить все вершины дерева на группы по последним K буквам на пути из корня в них, для каждого шаблона достаточно будет перебрать в среднем $\mathcal{O}(1)$ вершин, которые потенциально могут содержать вхождение этого шаблона, из соответствующей группы. Для проверки вхождения шаблона целиком также воспользуемся хешами.

В итоге полное решение работает за $\mathcal{O}(nK + m \cdot \frac{n}{26^K})$ в среднем.

Задача D. Творец воспоминаний

Автор задачи: Даниил Орешников, разработчик: Егор Юлин

Первая подгруппа решалась при помощи перебора. Давайте после каждого добавленного воспоминания проверять поле, и если надо, удалять строки. Каждую операцию будем выполнять за линейное время, если матрицу поля хранить построчно. Чтобы понять, куда «упадет» очередное воспоминание, достаточно знать самые верхние занятые клетки в соответствующих трех столбцах.

Во **второй подгруппе** нужно было действовать немного оптимальнее. Для каждой строки будем хранить количество клеток в этой строке, и если она становится в точности равной m, то удалять эту строку. Это необходимо, чтобы каждый раз заново не проверять, заполнена ли полностью строка. Поскольку необходимо хотя бы $\frac{m}{3}$ воспоминаний, чтобы удалить не более 3 строк, удаление строк будет происходить не чаще одного раза в $\frac{m}{9}$ запросов в среднем, что позволяет удалять строки явным образом, если быстро проверять необходимость удаления.

В **третьей и четвертой подгруппах** для начала заметим, что удаляться всегда будут только нижние строки, так как нет возможности заполнить верхние, не заполнив нижние. Одно из решений — будем хранить дерево отрезков по позициям $x_i \bmod 3 = 1$. Тогда добавление воспоминания — это прибавление 1 (или 3 в случае четвертой подгруппы) в точке. При этом удаление строк происходит, если минимальное значение в ДО больше 0 и соответствует вычитанию 1 на всем ДО. Можно было обойтись и без ДО в реализации, если заметить, что строки удаляются не так уж и часто (см. выше).

В **пятой подгруппе** воспоминания всегда падают на одни и те же непересекающиеся множества столбцов. Это позволяло разделить все столбцы на `frac`$m3$ независимых групп и работать с ними отдельно. Какие-то аспекты решения при этом могли быть реализованы проще, остальное решение не отличается от описанных ниже.

В **шестой подгруппе** из-за редкого удаления строк могло пройти аккуратно написанное простое решение из одной из первых групп. В качестве возможной оптимизации для каждого столбца будем хранить `std::set` — позиции текущих клеток. При удалении строки будем запоминать, что она удалена, в отдельном `std::set`, и при этом из всех множеств для столбцов удалять соответствующую клетку. Также будем увеличивать доступную высоту поля на 1, так как строка удаляется неявно и номера клеток не уменьшаются.

Тогда, чтобы узнать максимальную высоту в столбце, нам достаточно получить максимальную клетку в множестве для столбца, а затем вычесть количество удаленных ниже нее строк (бинпоиск в множестве номеров удаленных строк). Затем надо пропустить все номера уже удаленных строк и взять следующий свободный — это тоже можно сделать с помощью бинпоиска, если хранить множество всех еще неудаленных номеров строк.

Для **полного решения** будем в каждом столбце поддерживать декартово дерево, в котором будут храниться высоты всех его клеток. Тогда очистка ряда — это удаление соответствующих элементов из каждого столбца, и применение операции -1 ко всем позициям выше удаляемой строки (так как ряд сдвигается вниз). Удалять клетки можно просто вручную, так как удаленных клеток будет не больше, чем добавленных, то есть $9q$.

Для того, чтобы поддерживать сумму в каждой строке будем хранить ДД, где значением будет количество блоков в текущей строке. Тогда строку необходимо удалить, если значение равно m.

После чего номера более высоких строк можно уменьшить на 1 аналогичной отложенной операцией -1 ко всем ключам на поддереве. Суммарно такое решение работает за $\mathcal{O}(q \log n + q \log m)$.