Problem A. Anno Domini 2022

Идея: Дмитрий Штукенберг
Разработка: Дмитрий Штукенберг

Сопоставим годам нашей эры положительные числа, а годам до нашей эры — отрицательные, сдвинутые на 1 (1 год до н.э. будет отображаться в число 0):

$$Y_s = \begin{cases} \texttt{int } x, & s \equiv \text{“}\texttt{AD } x\text{”} \\ -\texttt{int } x + 1, & s \equiv \text{“}x \texttt{ BC}\text{”} \end{cases}$$

Тогда решение задачи сводится к вычислению модуля разницы между годами: $|Y_a - Y_b|$.

Альтернативный вариант — вычитать 1 из модуля разницы, если года принадлежат разным эрам, не изменяя самих значений годов:

$$|Y_a - Y_b| - (\texttt{if } Y_a \cdot Y_b > 0 \texttt{ then } 0 \texttt{ else } 1)$$

Problem B. Boris and Berta

Идея: Георгий Корнеев
Разработка: Георгий Корнеев

Переберём все возможные M в диапазоне $0..n/m+1$, где $+1$ требуется из-за того, что иногда выгодно промахнуться в большую сторону. Так как $n_{max}/m_{min} = 2 \cdot 10^6$, мы успеем это сделать.

Для каждого M вычислим оптимальный $C = \text{round}((n - m \cdot M)/c)$. В какую сторону при этом произойдёт округление для полуцелых чисел — не важно, так как итоговая погрешность будет одинаковой. Таким образом, проблем с точностью нет.

Выберем из всех сгенерированных пар (M, C) пару, приближающую n с наименьшей погрешностью.

Problem C. Clean Up!

Идея: Борис Минаев
Разработка: Борис Минаев

Для начала построим бор на строках из входа и будем решать задачу на получившемся дереве. Пометим вершины, в которых заканчиваются строки. Мы можем удалять поддерево, в котором осталось не более k помеченных вершин, за одну операцию. Понятно, что если некоторую вершину дерева нельзя удалить за одну операцию, то необходимо вначале удалить некоторое поддерево этой вершины.

Будем обрабатывать вершины от более глубоких и поддерживать инвариант, что в поддереве любой обработанной вершины осталось не более k помеченных вершин. Если суммарно в поддереве больше k таких вершин, выберем сына с наибольшим количеством помеченных вершин и удалим его за одну операцию. Будем удалять наибольшего сына, пока не окажется, что суммарное количество помеченных вершин в поддереве не превосходит k.

После того как все дерево было обработано, гарантируется, что осталось не более k помеченных вершин, так что их можно удалить за одну дополнительную операцию.

Problem D. Day Streak

Идея: Геннадий Короткевич
Разработка: Геннадий Короткевич

Изначально мы находимся в часовом поясе $t = 0$. Давайте поддерживать значения $d_i = \lfloor \frac{a_i + t}{m} \rfloor$ — дни, в которые решены задачи. Начнём увеличивать t вплоть до $m - 1$. Тогда в процессе каждое d_i не более одного раза увеличится на 1.

Будем поддерживать два упорядоченных множества. Множество B будет содержать все такие позиции i, что $d_{i+1} - d_i > 1$ — то есть все такие позиции, где обрывается цепочка подряд идущих дней с решёнными задачами. Множество S будет содержать длины всех имеющихся цепочек подряд идущих дней.

Тогда при увеличении d_i на единицу достаточно сделать $O(1)$ операций с множествами B и S, чтобы поддержать их в актуальном состоянии. В качестве структуры для множества можно использовать, например, `std::set` в C++ и его аналоги в других языках программирования.

Временная сложность решения составит $O(n \log n)$.

Альтернативно, можно использовать метод двух указателей, если научиться отвечать на вопрос «могут ли задачи $i, i+1, \ldots, j$ быть решены без пропусков дней». Для ответа на вопрос можно использовать дерево отрезков по часовым поясам, которое для каждого часового пояса будет хранить то, сколько пар задач $(k, k+1)$ образуют пропуск в этом поясе ($i \le k < j$). Тогда, если минимум по всему дереву отрезков равен 0, то существует часовой пояс, в котором пропусков нет, и можно двигать правый указатель, иначе нужно двигать левый указатель. Сложность такого решения также составит $O(n \log n)$.

Problem E. Extreme Problem

Идея: Максим Буздалов
Разработка: Максим Буздалов

Из-за того, что в задаче всего восемь возможных тестов (и на два из них в условии уже даны возможные правильные ответы), её решение сводится к предподсчёту ответов. При этом файл решения может фактически содержать большой `switch` по возможным вариантом входных данных. Остаётся понять, как можно организовать этот предподсчёт. Принципиально различных вариантов два: поиск ответа «на бумаге» и случайный поиск вариантов, причём к каждому тесту могут применяться разные варианты.

Случайный поиск вариантов — способ решения, который не требует значительного объёма размышлений, однако может потребовать достаточно большого объёма кода и некоторой веры в себя. Действительно, тот факт, что в этой задаче не бывает ответа «`No solution`», на первый взгляд неочевиден, а если неаккуратно написанный случайный поиск долго не находит ответа на какой-либо из тестов, возникает соблазн подумать, что ответа всё-таки нет.

Для реализации этого способа необходимо реализовать аналог чекера, способный определять наличие или отсутствие множественных локальных оптимумов и плато, а также проверять, для всех ли возможных пар (x, y) вычисления не переполняют 32-битный целый знаковый тип и вообще ведут себя корректно. Отметим, что часть проверок, например, проверку корректности использования стека, стоит возложить на генератор возможных функций, чтобы для чекера соответствующие условия выполнялись по построению, и на генерацию некорректных функций не тратилось бы время.

Остаётся реализовать генератор функций. Его можно реализовать различными способами. Работает, например, такой: фиксируется нечётная длина последовательности операций, далее каждый вид операции (константа, две переменных, четыре бинарных операции) генерируется равновероятно с учётом того, какие из операций вообще возможны при текущем размере стека и числе оставшихся операций, при этом значение для константы берётся равновероятно из интервала $[-9; 9]$. Для достаточной скорости поиска рекомендуется ограничить возможную длину последовательности каким-нибудь небольшим числом (в авторском решении используется число 17), а саму длину плавно увеличивать в ходе поиска.

При достаточной аккуратности реализации такое решение может найти ответ для каждого теста даже непосредственно в тестирующей системе, укладываясь в ограничения по времени. Авторское решение этой задачи устроено именно так.

Поиск ответа на бумаге, в силу достаточно свободного формата задания функции, может быть выполнен огромным множеством способов. Приведём один из них, учитывающий в явном виде то свойство, что $0^0 = 1$, а $0^{x^2} = 0$ для любого $x \neq 0$.

В качестве базовой функции будем использовать функцию $x + y$, которая не имеет ни локальных минимумов, ни локальных максимумов, ни плато.

Если от нас хотят множества локальных минимумов, создадим два локальных минимума в точках, например, $(-5, 0)$ и $(+5, 0)$ путём прибавления соответственно таких функций:

- $-9 \cdot 9 \cdot 0^{(x+5)^2+y^2}$;
- $-9 \cdot 9 \cdot 0^{(x-5)^2+y^2}$.

Для множества локальных максимумов аналогично выберем точки $(0, -5)$ и $(0, +5)$ и сменим знак у дополнительных функций. Наконец, для плато занулим значение функции в точке, к примеру, $(0, 1)$, добавив аналогичную конструкцию, вычитающую в этой точке единицу.

Problem F. First to Solve

Идея: Геннадий Короткевич
Разработка: Геннадий Короткевич

Разделим задачу на две части.

В первой части заполним матрицу $p(i, j, t)$ — вероятность того, что участник i решит задачу j ровно через t минут после начала соревнования.

Пусть список задач, которые участник i умеет решать, выглядит как $j_1, j_2, \ldots, j_s$. Зафиксируем конкретную задачу j из этого списка. Переберём, какой по счёту она окажется в списке участника i после случайной перестановки — обозначим эту позицию c. Рассмотрим также конкретный момент времени t.

В скольких перестановках задач $j_1, j_2, \ldots, j_s$ задача j окажется на c-м месте, и при этом окажется решена на минуте t?

Пусть $f(i, j, t, c)$ — число подмножеств задач $j_1, j_2, \ldots, j_s$ размера $c - 1$, не содержащих j, с суммой соответствующих элементов $a_{i,*}$ равной $t - a_{i,j}$. Можно заметить, что ответ на вопрос выше равен $f(i, j, t, c) \cdot (c - 1)! \cdot (s - c)!$.

Суммируя все количества перестановок по различным c и деля на общее число перестановок, чтобы перейти к вероятностям, получим формулу для $p(i, j, t)$:

$$p(i, j, t) = \sum_{c=1}^{s} f(i, j, t, c) \cdot (c - 1)! \cdot (s - c)!/s!.$$

Осталось научиться эффективно вычислять $f(i, j, t, c)$. Заметим, что перед нами вариация задачи о рюкзаке, которую можно решать с помощью динамического программирования. Кроме того, при фиксированном i для разных j задачи практически эквивалентны, и отличаются между собой только отсутствием одного предмета.

Для фиксированного i вычислим $g(c, t)$ — число подмножеств $a_{i,j_1}, \ldots, a_{i,j_s}$ размера c с суммой t — с помощью динамического программирования за $O(m^2k)$, добавляя все m «предметов» по одному, каждый за $O(mk)$. Теперь для каждого j давайте «откатим» добавление предмета $a_{i,j}$ — проделаем те же операции, что и при добавлении элемента, но в обратном порядке и с противоположными знаками. В этот момент мы получим матрицу с суммами подмножеств как раз всех элементов, кроме j-го, что и требовалось. Потом вернём предмет $a_{i,j}$ обратно. Так как удаление одного предмета работает за такое же время, как и добавление, все j будут обработаны в сумме за $O(m^2k)$.

Вспоминая, что мы фиксировали i, итоговое время работы всей первой части составит $O(nm^2k)$. При аккуратной реализации проходили по времени и менее эффективные алгоритмы — например, с асимптотиками $O(nm^2k \log m)$ и $O(nm^3k)$.

Во второй части задачи, используя матрицу $p(i, j, t)$, вычислим искомые ответы.

Зафиксируем задачу j и начнём идти по увеличению t. Для каждого участника i будем поддерживать вероятность q_i того, что он ещё не сдал задачу j, исходно равную 1. Тогда, чтобы обработать очередное t, нужно сделать следующее:

- для каждого участника i добавить к его ответу $p(i, j, t) \cdot \prod_{i_0 \neq i} q_{i_0}$;
- для каждого участника i уменьшить q_i на $p(i, j, t)$.

Чтобы быстро найти произведение q_{i_0} по всем $i \neq i_0$, можно сначала найти произведение всех значений q, а потом делить его на q_i для каждого i. Тогда время работы второй части решения составит $O(nmk)$.

Итоговая сложность решения — $O(nm^2k)$.

Problem G. Grand Center

Идея: Николай Карпов
Разработка: Павел Кунявский

Зафиксируем некоторое значение дисбаланса d. Рассмотрим некоторую точку x_0. Нужно получить условие, при котором ответ для этой точки не больше, чем d.

Переберем сторону, через которую прямая пересечет многоугольник с той стороны, где отрезок меньше, и рассмотрим некоторое направление. Опустим перпендикуляр из точки x_0, а также второй точки пересечения на сторону. Из подобия полученных треугольников следует, что чтобы дисбаланс для этого направления был не больше d, нужно, чтобы точка x_0 была не более чем в $d+1$ раз ближе к прямой, содержащей отрезок, чем вторая точка пересечения.

Заметим, что достаточно, чтобы это условие выполнялось для самой далекой от данной стороны вершины многоугольника, так как тогда оно будет выполнено и для всех остальных направлений. С другой стороны, выполнение этого условия для нее является необходимым, так как даже если прямая проходящая через нее и точку x_0 не пересекает сторону, то отрезок до второй точки пересечения будет только меньше, чем до точки пересечения с прямой содержащей сторону. Множество таких точек образует полуплоскость, паралельную стороне.

Таким образом необходимо и достаточно, чтобы точка лежала в пересчении n полуплоскостей. Для того, чтобы найти эти полуплоскости, достаточно для каждой стороны найти самую далекую от нее точку, и сдвинуть прямую, проходящую через сторону, на нужное расстояние. Это можно сделать за линейное время. Проверить непустоту пересечения полуплоскостей можно за время $O(n \log n)$ или же за линейное время рандомизированным алгоритмом. Значение же d можно подобрать бинарным поиском. При этом сортировать полуплоскости достаточно один раз, а пересечение уже отсортированных можно находить за линейное время. В таком случае общее время решения составит $O(n \log \frac{1}{\varepsilon})$. Решение с лишним $O(\log n)$ также можно написать, чтобы оно укладывалось в ограничения.

Интересным вопросом является верхняя граница для бинарного поиска. Из решения выше можно показать, что ответ всегда не больше, чем 2. Заметим, что для треугольника ответ всегда не больше (а на самом деле равен) 2 — подходит его центр масс. Это значит, что для $d = 2$ любые три полуплоскости в решении выше пересекутся. В таком случае, по теореме Хелли, и пересечение всех их вместе непусто, т. е. существует точка, ответ для которой не больше 2.

Кроме того, из доказательства решения следует, что множество точек, для которых ответ не больше чем d, является выпуклым. А значит, многие численные методы поиска минимума этой функции (например, два вложенных тернарных поиска) корректно работают. Осталось только научится эффективно вычислять функцию в точке. Для этого необходимо заметить, что худшим направлением всегда является направление через одну из вершин. А найти сторону, через которую пройдет прямая в направлении вершины, можно методом двух указателей для всех вершин за линейное время. Однако у такого решения могут быть трудности с точностью и/или временем работы. А также с

тем, что функция не определена снаружи многоугольника, что может быть неудобно для многих численных методов.

Problem H. Halfway There

Идея: Артем Васильев
Разработка: Артем Васильев

Ключевая идея решения состоит в том, что если число x взаимно просто с n, то и $n - x$ также взаимно с n. Это можно доказать, используя стандартное рекуррентное соотношение для алгоритма Евклида: $\gcd(n, n - x) = \gcd(n, -x) = \gcd(n, x) = 1$.

Это означает, что в списке взаимнопростых с n чисел для каждого числа $x < \frac{n}{2}$ есть парное число $n - x > \frac{n}{2}$. Также, для всех чисел, кроме $n = 2$, $\frac{n}{2}$ не принадлежит списку взаимно простых чисел. Это значит, что для $n > 2$ список имеет четную длину, и содержит одинаковое количество чисел, меньших и больших $\frac{n}{2}$. Задача свелась к тому, чтобы найти взаимно простое с n число, которое ближе всего к $\frac{n}{2}$ снизу.

Найти такое число можно простым циклом: начнем с $k = \lfloor \frac{n}{2} \rfloor$ и будем уменьшать k, пока $\gcd(n, k) > 1$. Интуитивно кажется, что много подряд чисел, имеющих общий делитель с n, быть не может, поэтому этот цикл быстро останавливается. Оценить сверху количество итераций можно с помощью расстояния между соседними простыми числами (англ. *prime gap*) или функции Якобсталя (англ. *Jacobstahl function*), равной максимальному расстоянию между взаимно простыми с n числами.

Однако, если копнуть глубже, можно увидеть, что ответом является одно из чисел $\lfloor \frac{n}{2} \rfloor$, $\lfloor \frac{n}{2} \rfloor - 1$ или $\lfloor \frac{n}{2} \rfloor - 2$. Разберем несколько случаев:

- Для нечетных чисел $n = 2k + 1$ ответ всегда равен k, поскольку $\gcd(2k + 1, k) = gcd(1, k) = 1$.
- Если $n = 4k$, то ответом будет $2k - 1 = \frac{n}{2} - 1$, потому что $\gcd(4k, 2k - 1) = \gcd(2, 2k - 1) = 1$, так как $2k - 1$ нечетно, а $2k$ четно.
- Если же n имеет остаток 2 по модулю 4, то есть, $n = 4k + 2$, то ответом будет $2k - 1$: $\gcd(4k + 2, 2k - 1) = \gcd(4, 2k - 1) = 1$, а $\frac{n}{2} = 2k + 1$ и $\frac{n}{2} - 1 = 2k$ не взаимно просты с n (кроме случая $n = 2$, который нужно рассмотреть отдельно).

Problem I. Imprecise Permutation Sort

Идея: Михаил Дворкин
Разработка: Михаил Дворкин

В этой задаче, хочется надеяться, есть несколько разных подходов, позволяющих уложиться в ограничение в 300 000 запросов, основанных на разных алгоритмах сортировки с надстройками для учёта странного компаратора. Опишем решение основанное на быстрой сортировке Хоара с элементами сортировки вставками, которое укладывается в ограничение «с запасом».

Но сперва исследуем, насколько вообще разрешима задача — можно ли отличить предложенными неточными сравнениями любое число из отрезка $[1, n]$ от другого. Несложная алгебра показывает, что с помощью какого-то меньшего числа b можно отличить числа a и $a + 1$ всегда кроме случая, когда a заканчиввается на 99. Говоря «можно отличить» мы имеем в виду, что $b =_{imprecise} a$ и при этом $b <_{imprecise} a + 1$. А пары неотличимые благодаря меньшим числам, к счастью, можно отличить благодаря какому-то большему числу c, то есть $c >_{imprecise} a$ и при этом $c =_{imprecise} a + 1$. Единственная фундаментальная проблема — если это большее число c не влезает в отрезок $[1, n]$.

Так при $n = 16\,384$ невозможно отличить числа 16 299 и 16 300. А при $n = 16\,300$ невозможно отличить не только их, но и 16 199 от 16 200. Впрочем, при любом n из предлоенного диапазона таких пар будет не более чем две, поэтому можно решить задачу, расставив эти пары как угодно, а затем за

2^2 операций обмена, перебирая все возможные расположения этих пар, добиться отсортированности массива.

Теперь заметим, что запросы на обмен местами (swap) тратят имеющийся ресурс в 300 000 запросов, поэтому минимизируем их количество. Для этого решим задачу сортировки индексов, не делая ни единого обмена, получим полное понимание (кроме одной—двух неотличимых пар), какой элемент где находится, и расположим их на свои места простым линейным алгоритмом за 16 383 обменов или меньше — например, будем жадно ставить любой элемент, лежащий не на своём месте, на своё место, пока такой вообще есть.

Затем заметим, что жюри может подготовить какой-нибудь неприятный для нашего решения тест. Чтобы лишить их такой возможности, сгенерируем произвольную перестановку π размера n и будем в запросах на сравнение каждый раз вместо пары (i, j) спрашивать о паре $(\pi(i), \pi(j))$. Тогда можно считать, что жюри даёт нам работать со случайной перестановкой и в ней нет никаких заготовленных «спецэффектов».

Эти идеи стоит применить в любом решении. Теперь опишем подход QuickSort+InsertionSort.

Для начала применим общую идею быстрой сортировки Хоара (принцип «разделяй и властвуй»), но получим как результат не отсортированный массив, увы, а набор групп элементов, таких что любой элемент более левой группы меньше любого элемента более правой.

Выберем произвольный элемент-разделитель (благодаря идее перестановки π можно выбрать, например, первый). Разделим за линейное время все элементы на меньшие, «равные» и большие его согласно неточному компаратору. Меньшие элементы обработаем этим же алгоритмом рекурсивно, большие — тоже, и положим полученные группы в ответ в естественном порядке: полученные рекурсивно группы с меньшими числами, группа чисел «равных» разделителю, и полученные рекурсивно группы с большими числами.

Кстати, первые 98 групп будут точно состоять из одного числа каждая, что помогает думать над этим решением, а также отлаживать его.

Теперь применим идею сортировки вставками. Инвариант будет такой: числа из первых нескольких групп уже идеально отсортированы кроме, возможно, одной или двух пар чисел вида $100k - 1$ и $100k$ (то есть неотличимых меньшими числами пар), которые могут быть перепутаны между собой. Причём речь идёт только о старших одной-двух парах, то есть тех, для которых различающее их большее число просто ещё не было добавлено в отсортированную область.

Научимся одну следующую группу вставлять в отсортированную область. Для этого к каждому элементу новой группы применим двоичный поиск, ищущий границу между числами меньшими его и числами «равными» ему. Переформулируем искомую величину так: сколько чисел в отсортированной области «равны» данному числу. И если найденная двоичным поиском граница оказалась вблизи неразличимых пар, то аккуратно уточним искомую величину, сделав ещё несколько сравнений.

Теперь все числа добавляемой группы разбились на две подгруппы: те, у которых хотя бы одно число в отсортированной области оказалось «равно» ему, и те, у которых таковых не нашлось. Первую группу уже можно добавлять к отсортированной области, а именно в порядке уменьшения количества чисел «равных» данному. А со второй группой следует повторить описанную процедуру ещё раз после добавления первой группы, ведь теперь отсортированная область немного выросла, и данных стало больше.

Кроме того, если после добавления очередной группы, то есть после нарастания отсортированной области, какую-то пару вида $100k - 1$ и $100k$ стало возможно отличить (добавилось различающее их большее число), лучше отличить их сейчас.

Итого, после добавления последней группы мы имеем отсортированный массив кроме одной-двух неразличимых пар. Как было сказано выше, теперь за не более чем $n - 1$ обмен расположим числа в массиве a в соответствии с нашим пониманием порядка и за не более чем 2^2 обмена перебором найдём расположение чисел в неразличимых парах.

Добавим также, что как и в «обычных» задачах о сортировке, QuickSort работает лучше, если элемент-разделитель выбирать не просто случайно, а как медиану из случайной выборки некоторого нечётного размера. В данной задаче это действительно помогает уменьшить число сравнений. А искать медиану небольшой нечётной выборки можно либо отсортировав эти числа по критерию «скольких чисел в этой выборке данное число меньше», либо ещё эффективнее: добавив числа из выборки в сбалансированное дерево поиска, используя неточный компаратор, но заменяя ответ «равно» на случайный ответ меньше/больше.

Также, в зависимости от реализации, может помочь уменьшить количество сравнений запоминание полученных ответов от компаратора: если про пару (i, j) уже спрашивали, то незачем неидеально написанному коду повторно спрашивать про (i, j) или про (j, i).

Программа с описанными здесь идеями делает на 1000 тестах от 272 185 до 287 010 запросов, в среднем 276 530 со стандартным отклонением 1425. Если вызывает опасения «хвост» этого распределения, то можно делать большего размера нечётные выборки для поиска медианного элемента-разделителя в быстрой сортировке, это немного стабильно увеличит число запросов, но уменьшит вероятность серии неудачных разделителей и превышения ограничения в 300 000 запросов.

Problem J. Journey in Fog

Идея: Дмитрий Саютин
Разработка: Геннадий Короткевич

Заметим, что стратегию Юлии можно описать числами $t_1, t_2, \ldots, t_n$ — моментами времени, когда произойдёт встреча, если Джейн двигается со скоростью $v_1, v_2, \ldots, v_n$, соответственно. При этом $t_1 > t_2 > \cdots > t_n$ — чем быстрее двигается Джейн, тем раньше произойдёт встреча.

На пары значений (t_i, t_{i+1}) можно наложить условия, вытекающие из ограничения скорости Юлии V. Но пока этого делать не будем, а посмотрим на то, как вычислить ответ по заданным $t_1, t_2, \ldots, t_n$.

Зафиксируем i. Встреча происходит в момент времени t_i, а после встречи Юлия потратит время $(L - v_i t_i)/V$ на дорогу домой. Сумма этих значений равна $(L + (V - v_i)t_i)/V$. Видим, что это линейная функция от t_i, причём если $v_i < V$, то коэффициент при t_i положительный, а если $v_i > V$, то отрицательный.

Найдём такое i_0, что v_{i_0} как можно ближе к V по абсолютному значению. Предположим, что мы зафиксировали t_{i_0}. Тогда все t_i для $i < i_0$ мы хотим минимизировать (поскольку коэффициент при них положительный), а все t_i для $i > i_0$ мы хотим максимизировать (поскольку коэффициент отрицательный). Неформально говоря, это значит, что от позиции, в которой Юлия будет находиться в момент времени t_{i_0}, ей нужно бежать в сторону Джейн с максимальной скоростью, если время течёт «вперёд», и бежать *от* Джейн с максимальной скоростью, если время течёт «назад».

Из этих рассуждений следует наблюдение о форме оптимального ответа: Юлия должна выбрать некоторое время x, которое она будет сидеть дома, а после этого бежать с максимальной скоростью в сторону Джейн, пока не встретит её.

Функция ответа от x является кусочно-линейной и состоит из n кусков. Можно в явном виде построить эту функцию и найти минимум по всех точкам излома, чтобы получить решение за $O(n)$. Также можно заметить или догадаться, что функция ответа от x унимодальна, и использовать тернарный поиск по x, получив решение за $O(n \log n)$.

Problem K. Kaleidoscopic Route

Идея: Михаил Дворкин
Разработка: Павел Маврин

Для начала давайте построим граф кратчайших путей из вершины 1 в вершину n. Для этого два раза запустим обход в ширину: из вершины 1 и из вершины n, и оставим в графе ребра (u, v), для которых $d(1, u) + d(n, v) + 1 = d(1, n)$, ориентировав их от u к v. Получившийся ациклический ориентированный граф содержит все кратчайшие пути из 1 в n, и только их (то есть любой путь

из 1 в n в этом графе — кратчайший). Теперь в этом графе нам нужно найти путь с максимальной разницей между максимальным и минимальным ребром.

Пусть кратчайший путь состоит из более чем одного ребра (если путь состоит из одного ребра, максимальная разница очевидно 0), тогда максимальное и минимальное ребро — это какие-то два различных ребра на этом пути. Пусть, например, мы сначала встретили минимальное ребро, а затем максимальное. Рассмотрим любую вершину на нашем пути между этими ребрами, пусть это вершина v, тогда путь делится на два отрезка: от 1 до v, и от v до n. Чтобы ответ был оптимальным, нужно чтобы на пути то 1 до v минимальное ребро было как можно меньше, а на пути от v до n максимальное ребро было как можно больше.

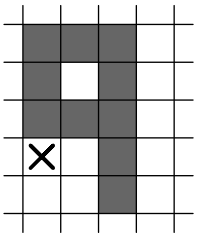
С помощью динамического программирования можно для каждой вершины v найти путь из 1 в v с наименьшим возможным минимальным ребром: $d_1[v] = \min(\min(d_1[u], c_{uv}))$. Аналогично можно найти для каждой вершины путь из v в n с наибольшим максимальным ребром: $d_2[v] = \max(\max(d_2[u], c_{vu}))$. Теперь найдем вершину с максимальной разностью $d_2[v] - d_1[v]$, и восстановим от нее соответствующие отрезки пути.

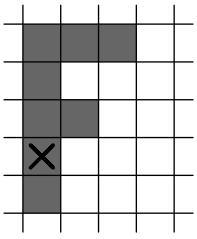
Аналогично найдем оптимальный путь, в котором сначала встречается максимальное ребро, а потом минимальное. Из этих двух ответов выберем наилучший.

Problem L. Letters Q and F

Идея: Николай Будин
Разработка: Николай Будин

Найдем самую левую, а среди таких — самую верхнюю, черную клетку. Это верхняя левая клетка либо буквы `Q`, либо буквы `F`. Посмотрим на клетку, находящуюся на 3 клетки ниже. Понятно, что если это буква `F`, то эта клетка будет черной. Несложно показать, что если это буква `Q`, то эта клетка будет белой (если бы она была черной, это бы означало, что изначально мы выбрали не самую левую клетку).





Таким образом, мы определили одну букву. Покрасим все клетки этой буквы обратно в белый цвет, и повторим процесс. Будем так делать, пока на поле остаются черные клетки. Заметим, что нет смысла каждый раз искать самую левую верхнюю клетку, начиная с начала. Вместо этого можно продолжить с того места, где мы нашли последнюю черную клетку. Таким образом, решение будет работать за $O(n \cdot m)$.

Problem M. Multithreaded Program

Идея: Виталий Аксёнов
Разработка: Виталий Аксёнов

Будем решать задачу с конца. Развернём все программы и будем искать те операции, которые приводят к финальному состоянию. Напишем следующий проход по всем программам: ищем либо операцию над переменной, которая приводит в нужное состояние, либо операцию над переменной, у которой уже выставлено значение. Выбранную операцию ставим в начало следующей в ответе (напомним, что программы сейчас развёрнуты). Если такой операции не существует, а программы ещё не закончились — ответ "No". Когда все программы выполнятся, можно выводить ответ.

За каждый проход мы обязательно выберем хотя бы одну операцию (можно выбрать несколько за раз). Время работы алгоритма получится $O(t \cdot \sum l_i)$.

Problem N. New White-Black Tree

Идея: Михаил Мирзаянов
Разработка: Нияз Нигматуллин

Утверждение 1: Дерево можно восстановить по степеням, если сумма степеней равна $2 \cdot (V-1)$ и каждая степень положительна.

Доказательство: по принципу Дирихле в таком списке степень будет хотя бы две единицы, то есть висячие вершины. Можно брать любую висячую вершину и подсоединять к вершине со степенью больше 1. В таком процессе степень суммарная уменьшается на два, одна вершина уничтожается и каждая степень оставшихся вершин положительна, то есть инвариант сохраняется. Такого сделать нельзя, когда осталось две висячие вершины, их можно просто соединить.

Утверждение 2: В дереве число висячих вершин — $2 + \sum\limits_{\deg(v)\geq 2} (\deg(v) - 2)$.

Доказательство: Пусть в дереве r висячих вершин. Тогда сумма степеней это $\sum \deg(v) = r + \sum\limits_{\deg(v)\geq 2} \deg(v)$, а сумма степеней это $2 \cdot (V-1)$. Получается, $2 \cdot (V-1) - r = \sum\limits_{\deg(v)\geq 2} \deg(v)$, вычтем из обеих частей $2 \cdot (V-r)$: $2 \cdot (V-1) - r - 2 \cdot (V-r) = \sum\limits_{\deg(v)\geq 2} \deg(v) - 2 \cdot (V-r)$. Упростим обе части: $2 \cdot (r-1) - r = r - 2 = \sum\limits_{\deg(v)\geq 2} (\deg(v) - 2)$. Получается, что $r = 2 + \sum\limits_{\deg(v)\geq 2} (\deg(v) - 2)$.

Для решения задачи проверим, что $\sum w_i + \sum b_i = 2 \cdot (n-1)$, $\sum w_i$ должна быть четной (из этих двух утверждений следует, что и $\sum b_i$ четна). Если мы сначала построим все белые ребра, то получим несколько компонент связности, а именно $k = \frac{\sum b_i}{2} + 1$ компонент связности. Потому что, если черными ребрами соединяем k компонент, то всего $k-1$ черных ребер, соответственно $\sum b_i = 2 \cdot (k-1)$.

По утверждению в начале доказательства, достаточно, чтобы в каждой полученной компоненте суммарная степень черных была положительна, так как сумма степеней четко связана с числом компонент.

Давайте возьмем k вершин с ненулевой черной степенью. Если это сделать нельзя, то решения нет. Дальше каждой вершине подсоединим нужное число белых висячих вершин. Это можно сделать, по утверждению 2, так как есть 2 висячие вершины + $\deg(v) - 2$ висячих для каждой вершины v в компоненте. Так мы построим k компонент, но некоторые вершины не используются. Их возьмем с $\deg(v) - 2$ висячими вершинами, удалим одно ребро из любой компоненты, и вставим эту вершину между ними, удовлетворив степень этой вершины, а у остальных не поменяв ничего.

После этого всего, осталось решить задачу для белых ребер, можно поступить как в доказательстве утверждения 1.

У задачи есть и более простые в реализации жадные решения. Например, работает стратегия, похожая на доказательство бесцветного утверждения 1: взять любую висячую вершину и подвесить ее к невисячей с максимальной степенью соответствующего цвета.