Технокубок 2019/2020

Отборочный раунд 1

1223A - КУС

Если n нечетно, то нам нужно докупить как минимум одну спичку, так как числа $a + b$ и c (a, b и c это элементы уравнения $a + b = c$) должны быть одной четности, а значит число $a + b + c$ всегда четно.

Если n четно, то мы можем собрать уравнение $1 + \frac{n-2}{2} = \frac{n}{2}$. Но есть один частный случай. Если $n = 2$, то нам нужно докупить две спички, так как числа a, b и c должны быть больше нуля.

Таким образом, ответ равен:

1. 2 если $n = 2$;
2. 1 если n нечетно;
3. 0 если n четно и больше чем 2.

1223B - Уравнивание строк

Если есть символ, который содержится в обоих строках s и t (обозначим это символ как c), то ответ «YES», так обе строки мы можем превратить в строку, состоящую только из символа c.

Иначе ответ «NO», так как если изначально строки не имели общий символ, то и после выполнения операций это не поменяется.

1223C - Спаси природу

Во-первых, будем считать, что $x \geq y$ (иначе, просто свапнем параметры программ). Пусть $cont(len)$ — максимальный вклад, который возможно получить, продав ровно len билетов.

Заметим, что в общем случае билеты могут быть одного из 4 типов:

- билеты, $(x + y)\%$ которых вкладывается в природу; предположим, их cXY штук;
- билеты, $x\%$ которых вкладывается в природу; предположим, их cX штук;
- билеты, $y\%$ которых вкладывается в природу; предположим, их cY штук;
- билеты, которые не попали ни в одну из программ.

Все значения cXY, cX и cY можно довольно просто посчитать, перебрав номер билета i от 1 по len и проверив, делится ли i на a или на b или на оба числа.

Теперь можно понять, что всегда выгодно выбрать в первую группу cXY максимальных элементов из p, во вторую группу — следующие cX максимумов и в третью группу — следующие cY максимумов.

Используя алгоритм выше мы можем посчитать значение $cont(len)$ за линейное время (отсортировать p можно заранее).

Последний шаг — это понять, что функция $cont(len)$ неубывающая, а потому мы можем подбирать len бинарным поиском, проверяя, что $cont(len) \geq k$.

Временная сложность алгоритма — $O(n \log n)$, но $O(n \log^2 n)$ также может проходить.

1223D - Сортировка последовательности

Давайте рассмотрим две последовательности чисел $m_1 < m_2 < \cdots < m_k$ и $d_1 < d_2 < \cdots < d_l$. Последовательность m содержит числа, которые были использованы в операциях в оптимальном ответе. Последовательность d содержит числа, которые не были использованы в таких операциях.

Например, если $a = [2, 1, 3, 5, 4]$, то оптимальный ответ будет следующим: переместить все 1-элементы в начало а затем переместить все 5-элементы в конец, таким образом $m = [1, 5]$ и $d = [2, 3, 4]$.

Для этих множеств выполняются два важных условия:

1. $maxInd(d_{i-1}) < minInd(d_i)$ для любых индексов i от 2 до l. $minInd(x)$ равно минимальному индексу i такому, что $a_i = x$, а $maxInd(x)$ равно максимальному индексу i такому что $a_i = x$;
2. для всех индексов i от 2 до l не существует числа x такого, что $d_i < x < d_{i+1}$ и последовательность m содержит число x.

Так как ответ в данном случае равен $|m| = k$, мы хотим минимизировать это значение. А значит мы хотим максимизировать длину последовательности d.

Таким образом, для индекса l мы хотим знать мксимальный индекс $dp_l = r$ такой, что мы можем отсортировать последовательность a без перемещений чисел из отрезка $l \dots r$. Это можно посчитать с помощью динамического программирования.

Для этого давайте пребирать все различные элементы последовательности a в порядке убывания $s_1, s_2, \dots, s_t$ ($s_{i-1} > s_i$ для всех i от 2 до t). Тогда если $maxInd(s_i) < minInd(s_{i+1})$ то $dp_i = dp_{i+1} + 1$, иначе $dp_i = 1$.

Ответ же равен $t - \max\limits_{i=1\dots t} dp_i$, где t это количество различных чисел в a.

1223E - Раскрась дерево

Очевидно, что если мы покрасим две вершины в один цвет, они должны быть смежными друг с другом, в противном случае мы могли бы покрасить их в разные цвета, и ответ был бы не хуже. Таким образом, мы можем свести задачу к следующей: выбрать набор ребер с максимальной стоимостью, чтобы ни одна вершина не была смежна с более чем k выбранными ребрами. Эта задача очень похожа на максимальное взвешенное паросочетание на дереве, и мы можем использовать некоторые методы, которые позволяют нам решить эту задачу.

Давайте решим задачу с помощью динамического программирования $dp_{v,f}$ — ответ на задачу для поддерева вершины v, f — это флаг, обозначающий, выбрано ли ребро между v и ее родителем. В зависимости от флага f мы можем выбрать k или $k-1$ ребер, соединяющие нашу вершину с ее детьми (обозначим максимальное количество ребер, которое мы можем выбрать, как t).

Мы должны выбрать не более t детей вершины v. Если мы выберем ребро, ведущее к вершине u, то к значению dp, которое мы сейчас вычисляем, будет добавлено $dp_{u,1} + w_{v,u}$, в противном случае будет добавлено $dp_{u,0}$.

Используя эту формулу, мы должны выбрать не более t детей вершины v, для которых сумма $dp_{u,1} + w_{v,u} - dp_{u,0}$ максимальна.

1223E - Раскрась дерево

Очевидно, что если мы покрасим две вершины в один цвет, они должны быть смежными друг с другом, в противном случае мы могли бы покрасить их в разные цвета, и ответ был бы не хуже. Таким образом, мы можем свести задачу к следующей: выбрать набор ребер с максимальной стоимостью, чтобы ни одна вершина не была смежна с более чем k выбранными ребрами. Эта задача очень похожа на максимальное взвешенное паросочетание на дереве, и мы можем использовать некоторые методы, которые позволяют нам решить эту задачу.

Давайте решим задачу с помощью динамического программирования $dp_{v,f}$ — ответ на задачу для поддерева вершины v, f — это флаг, обозначающий, выбрано ли ребро между v и ее родителем. В зависимости от флага f мы можем выбрать k или $k-1$ ребер, соединяющие нашу вершину с ее детьми (обозначим максимальное количество ребер, которое мы можем выбрать, как t).

Мы должны выбрать не более t детей вершины v. Если мы выберем ребро, ведущее к вершине u, то к значению dp, которое мы сейчас вычисляем, будет добавлено $dp_{u,1} + w_{v,u}$, в противном случае будет добавлено $dp_{u,0}$.

Используя эту формулу, мы должны выбрать не более t детей вершины v, для которых сумма $dp_{u,1} + w_{v,u} - dp_{u,0}$ максимальна.

1223F - Stack Exterminable Arrays

Let's understand how calculate the array nxt, such that nxt_l is equal to the minimum index $r > l$ such that subarray $a_{l \dots r}$ is stack exterminable. If there is no such index, then $nxt_l = -1$.

If we calculate this array then we solve this task by simple dynamic programming.

Let's calculate it in order $nxt_n, nxt_{n-1}, \dots, nxt_1$ by dynamic programming. At first consider simple case. If $a_i = a_{i+1}$, then $nxt_i = i + 1$. Otherwise we have to "add" the block $a_{i+1} \dots a_{nxt_{i+1}}$ (of course, nxt_{i+1} should be not equal to -1) and check that $a_i = a_{1+nxt_{i+1}}$. If this ($a_i = a_{1+nxt_{i+1}}$) also is not true, then you have to add a new block $a_{1+nxt_{i+1}} \dots a_{nxt_{1+nxt_{i+1}}}$ and check the condition $a_i = a_{1+nxt_{1+nxt_{i+1}}}$. If this condition also is not try, then you have to add a new block and so on.

It is correct solution, but it can be too slowly. Let's understand, that we add blocks to a_i until condition $a_i = a_{1+nxt_{\dots}}$ is holds. Let's assume, that we have an array $nxtX$ (this array contains a hashMaps, for example you can use map in `C++`), such that $nxtX_{i,x}$ is is equal to the minimum index $r > l$ such that subarray $a_{l \dots r}$ is stack exterminable and $x = a_{r+1}$. Then we can easily calculate the value $nxt_i = nxtX_{i+1,a_i} + 1$. Remains to understand, how to calculate $nxtX_i$. For this we just can make an assignment $nxtX_i = nxtX_{nxt_i+1}$. And then update $nxtX_{i,a_{nxt_i+1}} = nxt_i + 1$.

But I deceived you a little. We can't make an assignment $nxtX_i = nxtX_{nxt_i+1}$ because it is to slow. Instead that you need to swap elements $nxtX_i$ and $nxtX_{nxt_i+1}$, this can be done using the function $swap$ in `C++` or `Java` (time complexity of $swap$ if $O(1)$).

1223G - Деревянный плот

Давайте перебирать y от 2 по A, где $A = \max(a_i)$. И попытаемся найти лучший ответ для фиксированного y за время $O(\frac{A}{y})$.

Как это делать? Во-первых, мы может довольно просто посчитать суммарное количество бревен длины y, которые возможно получить (назовем это значение $cntY$): так как все $a_i \in [ky, ky + y)$ дают одинаковое количество бревен, равное k, то нам достаточно лишь посчитать количество таких a_i in $[ky, ky + y)$ для каждого k. Это можно сделать с помощью массива частот и префикс сумм на нем.

Есть два основных случая в задаче: оба бревна длины x отрезаны от одного бревна a_i или же от разных бревен a_i и a_j. Первый случай равносилен поиску одного бревна длины $2x$. Но в обоих случаях мы разобьем все возможные значения x ($2x$) на отрезки вида $[ky, ky + y)$ и проверим для каждого отрезка за $O(1)$.

Предположим, что $2x \in [ky, ky + y)$ и существует такое a_i, что $a_i \geq 2x$ и $(a_i \mod y) \geq (2x \mod y)$. В таком случае нам выгодно отрезать $2x$ от a_i и, более того, нам выгодно увеличивать $2x$, пока это возможно. Эти идеи наталкивают нас прямо на решение: будем поддерживать $\max(a_i \mod y)$ для $a_i \geq ky$ и проверять только $2x = ky + \max(a_i \mod y)$ (возможно, минус один, в случае неправильной четности). Мы можем поддерживать данный максимум перебирая k в порядке убывания. И, более того, $\max(a_i \mod y)$ для всех $a_i \in [ky, ky + y)$ это просто $\max(a_i \mid a_i < ky + y)$. Мы можем находить этот a_i за $O(1)$ с помощью предподсчета. Проверка выбранного $2x$ тривиальна: количество оставшихся бревен y равно $cntY - k$ и площадь возможного плота будет равна $y \cdot \min(x, cntY - k)$.

Случай с отрезанием x-в от разных a_i и a_j основан та той же идее, но требует поддерживание двух $mx1$ и $mx2$ ($mx1 \geq mx2$). Во в таком случае x может быть равен как $mx1$ or $mx2$.

Если $x = ky + mx2$, то все тривиально: количество бревен длины y равно $cntY - 2 \cdot k$ и так далее. Если же $x = ky + mx1$, то дополнительно необходимо проверить, что существуют необходимые $a_i \geq x$ и $a_j \geq x$. Тогда останется $cntY - 2 \cdot k - 1$ бревен длины y и так далее.

В результате, для каждого y мы умеем решать задачу за $O(\frac{A}{y})$, поэтому суммарная асимптотика равна $O(n + \sum_{y=2}^{A} O(\frac{A}{y})) = O(n + A \log A)$.

P.S.: Было решено позволить проходить решениям за $O(A \log^2 A)$, которые подбирают x бинарным поиском для каждого y, если они написаны аккуратно.

1240F - Football

Let's assume that $m \le n \cdot k$.

We can randomly assign colors to edges. If there is a vertex that does not satisfy the condition, then we can choose color a which appears the smallest number of times and color b which appears the biggest number of times. We will "recolor" edges that have one of these two colors.

Let's consider this graph only with edges with colors a and b. Let's add a "fake" vertex 0. This graph may have many components. If a component has at least one vertex with odd degree, we connect each such vertex with 0. Otherwise, let's choose any vertex from that component and add **two** edges to 0. Therefore, the graph will be connected and each vertex will have an even degree. Thus, we will be able to find an Euler cycle. Let's color the odd edges in the cycle in a, and even edges in b. As a result, the difference between these two colors for each vertex will be at most 1.

Let's do this operation while there is a vertex which does not satisfy the condition.

If $m > n \cdot k$, let's split the edges into two groups with the equal sizes (that is, $\lfloor \frac{m}{2} \rfloor$ and $\lceil \frac{m}{2} \rceil$). If a group has not greater than $n \cdot k$ edges, then do the algorithm at the beginning of the tutorial. Otherwise, split it again.

If you found the answers for two groups, you need to find the answer for the combined graph. Let f_1 be the most popular color in the first group, f_2 the second most popular color, ..., f_k the least popular color in the first group. Similarly, let g_1 be the most popular color in the second graph, etc. So, in the combined graph, f_1 should be equal to g_k, f_2 should be equal to g_{k-1}. In other words, we take the most popular color in the first group and color the least popular color in the second group with that color. If there is a vertex that does not satisfy the condition, "recolor" the graph according to the algorithm explained in the third paragraph.

Отборочный раунд 2

1225A - Забывчивость

Ответ существует только в случаях $d_a = d_b$, $d_b = d_a + 1$, или $d_a = 9$, $d_b = 1$. Также можно было просто проверить все значения a до 100 (или другой разумной границы).

1225B1 - Покупка подписок на сериалы (упрощённая версия)

Нам требуется найти подотрезок длины d с наименьшим числом различных значений. В маленьких ограничениях мы можем попробовать все такие подотрезки и посчитать количество различных элементов наивно (например, сортировкой или при помощи std::set).

1225B2 - Покупка подписок на сериалы (усложнённая версия)

В больших ограничениях нам потребуется использовать метод двух указателей, чтобы поддерживать количество различных элементов между подотрезками. Будем хранить map или массив, в котором будем считать количество вхождений каждого элемента, а также количество различных элементов (т.е. количество ненулевых значений в массиве). Чтобы подвинуть отрезок влево, нам нужно изменить два элемента массива и отследить, какие элементы в нем стали/перестали быть ненулевыми. Сложность $O(n \log n)$ или $O(n)$ (и то, и другое укладывается в ограничения).

1225C - p-двоичные числа

Пусть мы хотим представить n в виде суммы k p-двоичных чисел. Тогда мы хотим получить $n = \sum_{i=1}^{k} (2^{x_i} + p)$ для какого-то способа выбрать числа $x_1, \ldots, x_k$. Перенесём все p в левую часть: $n - kp = \sum_{i=1}^{k} 2^{x_i}$. В частности, это значит, что $n - kp$ должно быть не меньше k.

Посмотрим на двоичное представление $n - kp$. Если в нём больше k битов, равных 1, мы не сможем разбить это число на k степеней двойки. В противном случае возьмём двоичное представление, и если в нём меньше k степеней, мы всегда можем разбить большую степень на две меньших.

Попробуем все варианты k, начиная с маленьких. Если $n \ge 31p + 31$, то ответ не превосходит 31, поскольку $n - 31p$ меньше 2^{31}, а значит, оно всегда представимо в виде суммы 31 степени. В противном случае $n - 31(p + 1) < 0$. Раз $n > 0$, это значит, что $p + 1 < 0$, и $n - kp < k$ для всех $k > 31$, поэтому ответа нет.

1225D - Произведения-степени

Пускай $x \cdot y$ — k-я степень. Критерий для этого выглядит так: для любого простого числа p суммарное количество раз, которое оно делит x и y, должно делиться на k.

Разложим каждое число $a_i = p_1^{b_1} \dots p_m^{b_m}$ на простые множители, и сопоставим ему список пар $L_i((p_1, b_1 \bmod k), \dots, (p_m, b_m \bmod k))$, выкинув все пары, у которых $b_i \bmod k = 0$. Например, для $360 = 2^3 2^2 5^1$ и $k = 2$, список будет выглядеть так: $((2, 1), (5, 1))$.

Если $a_i \cdot a_j$ — k-я степень, то простые числа в их списках должны совпадать, а соответствующие $b_i \bmod k$ для одинаковых простых должны давать в сумме k. Действительно, если простое число отсутствует в одном из списков (т.е. показатель степени делится на k), то оно должно отсутствовать и в другом списке. В противном случае, суммарный показатель должен делиться на k, поэтому сумма остатков должна быть равна k.

Таким образом, для каждого a_i существует единственный список L'_i, который должен соответствовать второму числу, чтобы в произведении получилась k-я степень. Будем поддерживать количество раз, которое мы встретили каждый список (например, с помощью std::map), и для каждого числа будем прибавлять к ответу количество вхождений L'_i, а также увеличивать количество вхождений L_i на один.

Общая сложность решения состоит из факторизации всех чисел (за $O(\sqrt{max a_i})$ или за $O(\log \max a_i)$ с предподсчитанным решетом), а также поддержания map из векторов в числа. Суммарный размер всех векторов можно грубо оценить как $O(n \log n)$, поэтому суммарная сложность поддержания map составляет $O(n \log^2 n)$, либо $O(n \log n)$, если использовать хэш-таблицы.

1225E - Rock Is Push

Будем вычислять $R_{i,j}$ и $D_{i,j}$ – количество легальных способов дойти до цели, предполагая следующее:

- мы добрались до клетки (i, j);
- наш следующий шаг будет вправо/вниз соответственно;
- наш предыдущий шаг (если такой был) был *не в том же* направлении.

Положим $D_{n,m} = R_{n,m} = 1$ по определению.

Заметим, что в наших предположениях все камни, достижиые из (i, j), должны находиться на своих местах, поэтому ответ не зависит от того, как именно мы добрались до (i, j).

Пересчёт устроен довольно стандартно. Например, для $D_{i,j}$ пусть k — количество камней снизу от клетки (i, j). Мы сможем сделать не более $n - k - i$ шагов вниз, прежде чем повернём направо, поэтому $D_{i,j} = \sum_{t=1}^{n-k-i} R_{i+t,j}$. Это позволяет нам вычислять значения $R_{i,j}$ и $D_{i,j}$ при помощи динамического программирования, начиная с клеток с большими координатами.

Формула намекает на некий способ подсчёта сумм на отрезках, например, префиксные суммы или более сложную RSQ-структуру. Однако, в этой задаче можно обойтись без них. Действительно, если мы рассмотрим диапазоны суммирования для $D_{i,j}$ и $D_{i+1,j}$, мы заметим, что на каждом из концов они отличаются не более чем на один элемент. Поэтому для вычисления $D_{i,j}$ мы можем взять значение $D_{i,j}$ и прибавить/отнять не более двух значений $R_{i+t,j}$. Значения для клеток возле цели может понадобиться обработать отдельно, поскольку они не всегда реально являются суммами на отрезке. Также, нужно отдельно разобрать случай $n = m = 1$.

Общая сложность составляет $O(nm)$ (лишний $\log$ в решениях с RSQ не должен был сильно помешать).

1225F - Фабрика деревьев

Решим задачу с конца: превратим данное дерево в бамбук обратными операциями. В данном контексте, обратная операция выглядит так: для вершины v и двух её различных сыновей u и w, сделать w родителем u.

Какова нижняя оценка на количество операций, которые придётся сделать? Легко видеть, что *глубина* дерева, т.е. наибольшая длина пути от корня, может увеличиться не более чем на один за операцию. С другой стороны, глубина бамбука равна $n - 1$. Таким образом, нам понадобится как минимум $n - 1 - (\text{исходная глубина дерева})$ операций.

Нам хватило бы этого количества, если бы для любого дерева, не являющегося бамбуком, мы могли бы найти операцию, которая увеличивает его глубину. И это всегда возможно: посмотрим на самый длинный путь от корня. Если у каждой из вершин пути не более одного сына, то дерево — бамбук, и можно заканчивать. В противном случае, найдем любую вершину v на пути как минимум с двумя сыновьями, выберем её сына на пути в качестве u и любого другого сына в качестве w. Легко видеть, что после такой операции в дереве найдётся более длинный путь, чем раньше.

Находить все эти операции эффективно можно так: будем поддерживать указатель на самого низкого (далёкого от корня) кандидата для u. После применения операции, новый кандидат — это либо снова u, либо один из его предков. Пользуясь стандартным амортизационным анализом, можно показать, что все операции будут сделаны за время $O(n)$.

Чтобы восстановить ответ, выведем нумерацию в получившемся бамбуке, а потом сделанные операции в обратном порядке.

1225G - Получить 1

Опытный участник с ходу заметит решение динамическим программированием по подмножествам сложности примерно $O(3^n \sum a_i)$, но ограничения такое не позволяют. Что делать?

Пускай существует способ получить 1 в конце. Для каждого исходного числа a_i посчитаем, сколько раз это число в процессе поделится на k (включая деления чисел, содержащих a_i), из этого мы можем получить выражение $1 = \sum_{i=1}^{n} a_i k^{-b_i}$, где b_i — соответствующие количество делений. Что, если нам сразу дадут такое выражение? Сможем ли мы по нему восстановить необходимую последовательность операций?

Оказывается, сможем! Докажем по индукции. Если у нас всего одно число, оно должно быть 1 и мы победили. В противном случае, пусть $B = \max b_i$. Покажем, что есть как минимум два числа a_i и a_j, для которых $b_i = b_j = B$. Действительно, пусть есть всего одно такое число a_j. Домножим обе части выражения на k^B, получим $k^B = \sum_{i=1}^{n} a_i k^{B-b_i}$. Левая часть, а также все слагаемые правой части, кроме одного, делятся на k. Но a_j не делится на k, поэтому и вся правая часть не делится на k, противоречие.

Теперь, раз у нас есть два таких числа a_i и a_j, заменим их на $f(a_i + a_j) = a'$, и положим соответствующее $b' = B - (\text{количество раз, которое } (a_i + a_j) \text{ делится на } k)$. Видно, что новый набор чисел все ещё удовлетворяет выражению выше, поэтому по индукции мы сможем закончить цепочку преобразований. Заметим также, что это доказательство можно превратить в процедуру восстановления ответа: если даны числа b_i, сгруппируем любую пару a_i и a_j, для которых $b_i = b_j$.

Теперь задача в следующем: можем ли мы подобрать числа $b_1, \dots, b_n$, чтобы выражение выше было верным? Это можно сделать более аккуратным динамическим программированием по подмножествам. Для множества $S \subseteq \{1, \dots, n\}$ и числа x, пусть $dp_{S,x} = 1$ тогда и только тогда, когда $x = \sum_{i \in S} a_i k^{-b_i}$ для какого-то выбора b_i. Исходное состояние — это $dp_{\varnothing,0} = 1$. Переходы следующие:

- включить a_i в S: $dp_{S,x} \implies dp_{S+a_i,x+a_i}$;
- увеличить все b_i на 1: если x делится на k, то $dp_{S,x} \implies dp_{S,x/k}$.

По определению, мы сможем получить 1 тогда и только тогда, когда $dp_{\{1,\dots,n\},1}$. Вычисление значений dp напрямую приводит к сложности $O(n2^n \cdot \sum a_i)$. Однако, первый тип переходов можно соптимизировать битсетами, и сложность становится $O(2^n \sum a_i \cdot (1 + n/w))$, где w — длина машинного слова (32 или 64). Чтобы восстановить b_i, пройдём от $dp_{\{1,\dots,n\},1}$ обратно к $dp_{\varnothing,0}$ по достижимым состояниям.

1246F - Курсор и расстояния

Попробуем снова решить задачу с конца. Для конкретного символа s_i, из каких символов s_j в него можно попасть за один шаг? Легко видеть, что s_j должны лежать в подотрезке s, ограниченном ближайшими вхождениями символа s_i слева и справа (включительно), либо границами строки s. Для данного s_i, обозначим этот подотрезок $[L_i, R_i)$ (для удобства мы будем использовать полуинтервалы, *чего и вам желаем*).

В какие символы можно попасть из s_i за k обратных переходов? Видно, что такие символы также образуют подотрезок, обозначим его $[L_i^{(k)}, R_i^{(k)})$. Для этих подотрезков верно, что $[L_i^{(0)}, R_i^{(0)}) = [i, i+1)$, и $[L_i^{(k)}, R_i^{(k)}) = \cup_{j \in [L_i^{(k-1)}, R_i^{(k-1)})} [L_j, R_j)$.

Отметим, что по значениям концов $[L_i^{(k)}, R_i^{(k)})$ можно непосредственно вычислить ответ, например, по формуле $\sum_{i=1}^{n} \sum_{k=0}^{n-1} (n - (R_i^{(k)} - L_i^{(k)}))$. Эта формула верна, поскольку выражение в скобках равно количеству символов, в которые *нельзя* попасть из i за k обратных шагов, поэтому любая позиция j на расстоянии d учтётся ровно d раз. Если чуть-чуть упростить, получим формулу $n^3 - \sum_{i=1}^{n} \sum_{k=0}^{n-1} (R_i^{(k)} - L_i^{(k)})$.

Значения $[L_i^{(k)}, R_i^{(k)})$ можно найти за $O(n^2)$ по одному, если применить реккуренты выше и разреженные таблицы, но это всё ещё очень долго. Хочется применить двоичные подъемы, но с ростом k изменения $L_i^{(k)}$ и $R_i^{(k)}$ зависят друг от друга, а хранить таблицу размера n^2 мы не хотим. Тем не менее, с определённого момента они всё же начинают меняться независимо, а именно, когда диапазон $[L_i^{(k)}, R_i^{(k)})$ содержит все различные буквы, которые встречаются в s. Действительно, мы можем найти $R_i^{(k+1)}$ как наибольшее R_j, где j пробегает по последним вхождениям букв `a`, …, `z` слева от $R_i^{(k)}$. Немного обобщая, изменения $L_i^{(k)}$ и $R_i^{(k)}$ можно рассматривать независимо, пока количество различных символов в диапазоне не меняется. А измениться оно может не более $\alpha = 26$ раз...

Попробуем собрать всё вместе. Для каждого i будем хранить k_i — сколько раз мы расширили $[L_i^{(k)}, R_i^{(k)})$, а также сами границы диапазона. Будем увеличивать параметр t — количество различных символов в диапазонах, которые мы сейчас хотим расширить. Для каждого правого конца r посчитаем $f_r(r)$ — наибольшее R_j среди t ближайших уникальных букв слева от r; аналогично определим и посчитаем $f_l(l)$. Вместе с этим, посчитаем двоичные подъёмы $f_r^{2^y}(r)$, $f_l^{2^y}(l)$, и ещё для каждого двоичного подъёма сумму r или l, которые он пробегает. Теперь для каждого i двоичными подъемами найдём наибольшее количество расширений, которые мы можем применить к $[L_i^{(k)}, R_i^{(k)})$, чтобы количество различных символов в этом диапазоне всегда было равно t. Таким образом, мы успешно обновим k_i, концы $[L_i^{(k)}, R_i^{(k)})$, а также вычислим какую-то часть $\sum_{i=1}^{n} \sum_{k=0}^{n-1} (R_i^{(k)} - L_i^{(k)})$. И переходя в следующую фазу $t+1$, мы знаем всю необходимую в дальнейшем информацию.

Сколько это работает? Для каждого значения $t = 1, \ldots, \alpha = 26$ нам понадобится $O(n \log n)$ времени на построение и использование двоичных подъёмов для обновления всего, что нам нужно. Итоговая сложность получается $O(\alpha n \log n)$.

Отборочный раунд 3

1262A - Математическая задача

Найдём наиболее левую из всех правых границ отрезков, назовём ее r_{min}. Аналогично, найдём наиболее правую из всех левых границ отрезков, назовём ее l_{max}. Несложно заметить, что ответ на задачу равен $\max(0, l_{max} - r_{min})$.

1262B - Коробка

Очевидно, что если $q_i \neq q_{i-1}$, то $p_i = q_i$. Здесь мы считаем, что $q_0 = 0$. Все остальные позиции в перестановке могут быть заполнены оставшимися числами в порядке возрастания. После этого необходимо проверить удовлетворяет ли полученная перестановка условию задачи.

1261A - Грязно

Давайте найдем любую правильную скобочную последовательность, которая является ответом. Например, это может быть такая последовательность: $t =$ «`()()()()` … `(((...(()...)))))`».

Теперь преобразуем s в t за не более чем n операций.

Будем идти слева направо и для каждой позиции i, если:

- $s_i = t_i$ тогда ничего делать (менять) не надо,
- в противном случае, надо найти такую позицию j, которая $j > i$ и при этом $s_j = t_i$ (такая обязательно существует), затем перевернуть отрезок $[i \ldots j]$.

Количество операций не будет превосходить n, такое решение будет работать за $O(n^2)$.

1261B2 - Оптимальные подпоследовательности (усложненная версия)

Давайте сначала решим упрощенную версию, не обращая внимание на эффективность алгоритма. Понятно, что сумма элементов оптимальной подпоследовательности равна сумме k максимальных элементов последовательности a. Пусть наименьший (k-й) из k максимальных элементов равен x. Очевидно, что в ответ попадут все элементы a, которые больше x и несколько элементов, которые равны x. Среди всех элементов, которые равны x надо выбрать такие, которые расположены как можно левее. Таким образом, решение упрощённой версии может выглядеть так:

- чтобы построить оптимальную подпоследовательность длины k возьмем массив a b построим его отсортированную по невозрастанию копию b: $b_1 \geq b_2 \geq \cdots \geq b_n$;
- пусть $x = b_k$;
- возьмем из a следующую подпоследовательность: все элементы $a_i > x$ и самые левые вхождения $a_i = x$ (таких вхождений надо взять столько, чтобы суммарно получить ровно k элементов в ответе).

Чтобы решить усложнённую версию, заметим, что решение выше эквивалентно сортировки всех элементов a в первую очередь по значению (по невозрастанию), а во вторую очередь по позиции (по возрастанию). Искомая оптимальная подпоследовательность — это просто k первых элементов a в таком порядке.

Чтобы быстро отвечать на запросы воспользуемся возможностью считать в программе все запросы, отсортировать их по k_j и обрабатывать именно в таком порядке. Тогда чтобы ответить на запрос k_j, pos_j надо взять в таком порядке k_j элементов и выбрать из них pos_j-й (уже просто в порядке индексов). Таким образом, задача сводится к нахождению pos-го элемента в некотором множестве, куда только добавляются элементы. Это может быть решено с использованием широкого круга структур данных (дерево отрезков, дерево Фенвика, даже sqrt-композициями), так и с использованием полу-стандартного встроенного в g++ дерева, которое поддерживает операцию «найти быстро pos-й элемент множества». Ниже предложен код решения:

```cpp
#include <bits/stdc++.h>

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace std;
using namespace __gnu_pbds;

#define forn(i, n) for (int i = 0; i < int(n); i++)

typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> ordered_set;

int main() {
    int n;
    cin >> n;

    vector<int> a(n);
    vector<pair<int,int>> b;
    forn(i, n) {
        cin >> a[i];
        b.push_back({-a[i], i});
    }
    sort(b.begin(), b.end());

    int m;
    cin >> m;
    vector<pair<pair<int,int>,int>> req(m);
    forn(i, m) {
        cin >> req[i].first.first >> req[i].first.second;
        req[i].second = i;
    }
    sort(req.begin(), req.end());

    vector<int> ans(m);
    ordered_set pos;
    int len = 0;
    forn(i, m) {
        while (len < req[i].first.first)
            pos.insert(b[len++].second);
        ans[req[i].second] = a[*pos.find_by_order(req[i].first.second - 1)];
    }

    forn(i, m)
        cout << ans[i] << endl;
}
```

1261C - Поджог в лесу Берляндии

Заметим, что если существует конфигурация, при которой лес мог гореть T минут, то и существует конфигурация при которой лес мог гореть $T-1$ минут. А потому мы можем искать ответ с помощью двоичного (бинарного) поиска.

Теперь надо уметь проверять существование конфигурации для фиксированного T. Давайте найдем все возможные подожженные деревья. Есть два эквивалентных условия для таких клеток: либо квадрат размера $2T+1$ с центром в данной клетке состоит только из `X`-в, либо расстояние от данной клетки до любой «`.`» (или границы) не менее T.

Мы можем использовать любое из условий. Первый случай может быть проверен с помощью префикс-сумм в 2D - их можно предпосчитать один раз, а потом использовать сумму на прямоугольнике. Во втором случае мы можем запустить bfs из всех «`.`»-к или границ (или из всех `X`-в, граничащих с «`.`» или границей) также один раз перед бинарным поиском.

Второй шаг — это проверить, что все сгоревшие деревья действительно могли сгореть, если пожар начался с выбранных клеток. Мы можем это проверить с помощью «прибавления на прямоугольник» (с помощью префикс-сумм), так как каждая подожженная клетка сожжет квадрат $(2T+1) \times (2T+1)$ с центром в себе. Или же можно запустить bfs из всех подожженных клеток.

В любом случае, оба способа имеют сложность $O(nm)$. И, так как все уничтоженные деревья изображены на карте, ответ не может превосходить $\min(n, m)$. А потому, общая асимптотика равна $O(nm \log(\min(n, m)))$.

1261D2 - Неправильный ответ на тесте 233 (усложненная версия)

Для начала, необходимо обработать отдельно случай $k = 1$, при котором ответ равен нулю.

Обозначим за d разницу баллов у нового и старого ответа. Корректный набор ответов означает $d > 0$.

Для позиций, что $h_i = h_{i \mod n+1}$, ответ не влияет на d. Предположим, что есть t позиций, что $h_i \neq h_{i \mod n+1}$.

Для фиксированной позиции i, что $h_i \neq h_{i \mod n+1}$, пусть ваш ответ равен a_i. если $a_i = h_i$, тогда значение d уменьшится на 1. Назовем такие позиции позициями **уменьшения**. Если $a_i = h_{i \mod n+1}$, тогда значение d увеличится на 1. Назовем такие позиции позициями **увеличения**. В противном случае значение d не будет изменено, назовем такие позиции **нулевыми позициями**.

Очевидно, количество позиций увеличения должно быть строго больше чем количество позиций уменьшения. Переберем это количество нулевых позиций. Тогда ответ равен $k^{n-t} \times \sum_{0 \le i \le t-1} [(k-2)^i \times \binom{t}{i} \times \sum_{[\frac{t-i}{2}]+1 \le j \le t-i} \binom{t-i}{j}]$. i означает число нулевых позиций и j означает число позиций увеличения.

Единственная проблема в том, чтобы посчитать $\sum_{[\frac{t-i}{2}]+1 \le j \le t-i} \binom{t-i}{j}$ быстро. Так как $\binom{n}{x} = \binom{n}{n-x}$, можно заметить, что если $t-i$ нечетно, $\sum_{[\frac{t-i}{2}]+1 \le j \le t-i} \binom{t-i}{j} = 2^{t-i-1}$. В противном случае это равно $\frac{2^{t-i} - \binom{t-i}{\frac{t-i}{2}}}{2}$.

1261E - Не равны

Авторское решение было достаточно сложным, вы можете прочитать его в английской версии.

Рассмотрим более простое решение.

Отсортируем числа по невозрастанию, будем поддерживать классы операций с равными «`векторами`». После первого числа у нас будет два класса, первые a_i с единицей, и остальные $n+1-a_i$ с нулями. Далее для каждого числа рассмотрим любой класс в котором есть хотя бы два числа, поставим в одно из них в i-ю позицию ноль, а в другое туда же один. Остальные $a_i - 1$ можно расставить произвольно в оставшиеся вектора.

Можно заметить, что каждый раз количество классов увеличивается хотя бы на один (если их и так не $n+1$, конечно).

Таким образом будет ≥ 2 (исходно) + $(n-1) = (n+1)$ классов в итоге, получается все вектора в ответе будут различными.

1261F - Xor-множество

Рассмотрим дерево отрезков на отрезке $[0, 2^{60} - 1]$, вершина представляющая отрезок длины 2^n будет соответствовать числам у которых первые $60 - n$ биты совпадают со всеми возможными последними n битами. Иначе говоря двоичная запись чисел будет равна $a_1 a_2 a_3 \dots a_{59-n} a_{60-n} x_1 x_2 x_3 \dots x_n$, где все a_i и x_i это 0 или 1. Числа от a_1 до a_{60-n} фиксированны и x_i могут быть выбраны произвольным образом.

Можно заметить, что если у нас есть два отрезка A и B в дереве, все возможные числа которые получаются как числод из A и число из B также образуют отрезок в дереве, с длиной $\max(|A|, |B|)$, а неизменяемые биты также образуют неизменяемые биты, равные ксору неизменяемых бит двух отрезков.

Используя это наблюдение мы можем получить алгоритм за $O((n^2 \log^2 10^{18}) \log(n^2 \log^2 10^{18}))$ или $O(n^2 \log^2 10^{18})$, в зависимости от метода сортировки. Для начала, мы получим все отрезки которые получаются интервалами из A и B, мы получим $O(n^2 \log^2 10^{18})$ отрезков в результате. Затем нам нужно посчитать сумму в объединении этих отрезков. Можно отсортировать эти отрезки, но к сожалению это получит ML, так как отрезков порядка 10^8.

Чтобы соптимизировать этот алгоритм, можно заметить, что когда мы объединяем два отрезка описанным способом, меньший отрезок равен предку того же размера как больший отрезок. Назовем все отрезки декомпозиции «`настоящими отрезками`», а все отрезки с «`настоящими отрезками`» в поддереве «`вспомогательными`». Затем можно перебрать все 60 возможных значений размеров отрезка, и для каждого значения можно перебрать «`настоящий`» отрезок множества A и «`вспомогательный`» отрезок множества B и добавить значения в множество. Можно показать, что количество вспомогательных и настоящих отрезков каждой длины не превосходит $4n$. Таким образом, решение работает за $O((n^2 \log 10^{18}) \log(n^2 \log 10^{18}))$ или $O(n^2 \log 10^{18})$, в зависимости от метода сортировки.

Отборочный раунд 4

1259A - С днём рождения, Поликарп!

Наверное, один из самых простых способов решить эту задачу состоит в переборе всевозможных красивых чисел до 10^9 и проверки каждого из чисел на то, что оно не превосходит n. Вы можете перебирать в первую очередь длину от 1 до 8, поддерживая число вида `11...1` этой длины, внутри перебирать множитель для этого числа от 1 до 9. Основная часть такого решения может выглядеть примерно так:

```
cin >> n;
int b = 0, ans = 0;
for (int len = 1; len <= 9; len++) {
    b = b * 10 + 1;
    for (int m = 1; m <= 9; m++)
        if (b * m <= n)
            ans++;
}
cout << ans << endl;
```

1259B - Сделай нечетными

Рассмотрим максимальное положительное значение в наборе. Очевидно, что его придётся когда-нибудь поделить на два. Чем раньше это будет сделано — тем лучше (ведь полученные половины могут быть на более поздних шагах поделены надвое).

Таким образом, оптимальный метод для решения этой задачи такой: *до тех пор пока в наборе есть хотя бы одно чётное число, надо выбрать максимальное из таких значений и поделить все равные ему элементы на* 2.

Для эффективной реализации вы можете использовать возможности стандартной библиотеки для представления множества в виде `std::set` (это в C++, в других языках есть аналоги или можно модифицировать решение).

Ниже приведён пример возможной реализации основной части решения:

```
cin >> n;
set<int> a;
for (int i = 0; i < n; i++) {
    int elem;
    cin >> elem;
    a.insert(elem);
}
int result = 0;
while (!a.empty()) {
    int m = *a.rbegin();
    a.erase(m);
    if (m % 2 == 0) {
        result++;
        a.insert(m / 2);
    }
}
cout << result << endl;
```

1259C - Просто как one и two

Рассмотрим каждое из вхождений подстрок `one` и `two`. Очевидно, что в каждой такой подстроке надо удалить не менее одного символа. Эти подстроки никак не могут пересекаться, кроме одного случая: `twone`. Таким образом, ответ обязательно не меньше величины $c_{21} + c_1 + c_2$, где c_{21} — количество вхождений вида `twone`, c_1 — количество вхождений вида `one` (которые не являются частью `twone`), c_2 — количество вхождений вида `two` (которые не являются частью `twone`).

Предложим способ, который делает ровно $c_{21} + c_1 + c_2$ удалений и, таким образом, является оптимальным:

- В каждом вхождении `twone` удалим букву `o` (это «убьет» вхождение и `one` и `two`);
- Затем в каждом вхождении `one` удалим букву `n` (это «убьет» вхождение `one`);
- Затем в каждом вхождении `two` удалим букву `w` (это «убьет» вхождение `two`).

Заметим, что это принципиально важно в последних двух пунктах удалять средние буквы, чтобы после схлопывания строки после удаления не появилось нового вхождения.

Ниже приведён пример возможной реализации основной части такого решения:

```
string s;
cin >> s;
vector<int> r;
for (string t: {"twone", "one", "two"}) {
    for (size_t pos = 0; (pos = s.find(t, pos)) != string::npos;) {
        s[pos + t.length() / 2] = '?';
        r.push_back(pos + t.length() / 2);
    }
}
cout << r.size() << endl;
for (auto rr: r)
    cout << rr + 1 << " ";
cout << endl;
```

1259D - Сыграем в слова?

Для заданного набор строк несложно составить критерий того, что для него существует корректный порядок расположения слов при игре в слова (будем называть такие наборы строк *корректными*). Набор строк является корректным, если количество строк вида `0...1` и строк вида `1...0` отличается не более чем на 1 (в любую сторону) и при этом если существуют строки вида и `0...0`, и `1...1`, значит есть хотя бы одна строка вида `0...1` или `1...0`.

Это утверждение легко доказать, поняв что задача эквивалентна эйлерову обходу орграфа из двух вершин. Можно и доказать этот факт, не прибегая к теории графов:

- если есть и `0...0` и `1...1`, но нет `0...1` и нет `1...0`, то начав со строки одного вида нельзя перейти на строку другого вида (следовательно, *если существуют строки вида и `0...0` и `1...1` есть хотя бы одна строка вида `0...1` или `1...0`* — необходимое условие);
- если количество строк вида `0...1` и строк вида `1...0` отличается не более чем на 1 (в любую сторону), то можно называть их по очереди, начав с такого типа, которого больше (если одинаково — с любого, строки вид `0...0` и `1...1` можно подсовывать в любые подходящие моменты).

Перевороты влияют только на взаимное количество строк вида `0...1` и `1...0`. Поэтому сразу во время чтения входных данных можно проверить необходимое условие (первый пункт из перечисления выше).

Пусть без уменьшения общности количество строк вида `0...1` равно n_{01} и строк вида `1...0` — равно n_{10}. При этом пусть $n_{01} > n_{10} + 1$. Помним, что в текущем наборе все строки различны. Покажем, что можно всегда выбрать некоторые строки вида `0...1` и перевернуть так, чтобы стало $n_{01} = n_{10} + 1$ (и при этом все строки остались различными).

В самом деле, в наборе строк вида `0...1` не более n_{10} таких строк, что после переворота получится дубликат (потому что будет получатся строка вида `1...0`, а их всего только n_{10}). Значит есть не менее $n_{01} - n_{10}$ строк, которые можно перевернуть и дубликат при этом не образуется. Мы можем выбрать любые $n_{01} - n_{10} - 1$ из них.

Таким образом, после проверки необходимого условия (первый пункт из перечисления выше) надо перевернуть требуемое число строк из типа, которого слишком много, выбрав такие строки, переворот которых не приводит к появлению дубликатов.

Ниже приведен пример возможной реализации основной части описанного выше решения.

```cpp
int n;
cin >> n;
vector<string> s(n);
set<string> s01;
set<string> s10;
vector<bool> u(2);
forn(i, n) {
    cin >> s[i];
    if (s[i][0] == '0' && s[i].back() == '1')
        s01.insert(s[i]);
    if (s[i][0] == '1' && s[i].back() == '0')
        s10.insert(s[i]);
    u[s[i][0] - '0'] = u[s[i].back() - '0'] = true;
}
if (u[0] && u[1] && s01.size() == 0 && s10.size() == 0) {
    cout << -1 << endl;
    continue;
}
vector<int> rev;
if (s01.size() > s10.size() + 1) {
    forn(i, n)
        if (s[i][0] == '0' && s[i].back() == '1') {
            string ss(s[i]);
            reverse(ss.begin(), ss.end());
            if (s10.count(ss) == 0)
                rev.push_back(i);
        }
} else if (s10.size() > s01.size() + 1) {
    forn(i, n)
        if (s[i][0] == '1' && s[i].back() == '0') {
            string ss(s[i]);
            reverse(ss.begin(), ss.end());
            if (s01.count(ss) == 0)
                rev.push_back(i);
        }
}
int ans = max(0, (int(max(s01.size(), s10.size())) - int(min(s01.size(), s10.size())))) / 2);
cout << ans << endl;
forn(i, ans)
    cout << rev[i] + 1 << " ";
cout << endl;
```

1259E - Две ярмарки

Эта задача имеет простое линейное решение (просто два поиска в глубину) без привлечения точек сочленения, компонент двусвязности и прочих продвинутых техник.

Переформулируем эту задачу на языке теории графов: задан неориентированный граф и две вершины в нём a и b, найдите количество пар вершин (x, y), что любой путь из x в y содержит обе вершины a и b.

Иными словами, нам интересны такие пары вершин (x, y), что удаление вершины a (при этом оставляем b) нарушает связность от x до y и удаление вершины b (при этом оставляем a) нарушает связность от x до y.

Удалим вершину a и выделим компоненты связности в получившимся графе. Аналогично, удалим вершину b и выделим компоненты связности в получившимся графе. Тогда пара (x, y) нам интересна, если x и y принадлежат разным компонентам и при удалении a и при удалении b.

Таким образом, найдем для каждой вершины u пару (α_u, β_u) — номера компонент связности при удалении a и b, соответственно. Пара (x, y) нам интересна, если $(\alpha_x, \beta_x) \neq (\alpha_y, \beta_y)$.

Общее количество пар вершин равно $n \cdot (n-1)/2$. Вычтем из него количество неинтересных пар.

Во-первых, это такие пары, что (α_x, β_x) и (α_y, β_y) совпадают частично (по ровно одной компоненте). Например, пусть совпадение будет по первой компоненте по общему значению α. Пусть суммарное количество пар (α_u, β_u), что $\alpha_u = \alpha$ равно c, тогда вычтем из текущего ответа $c \cdot (c-1)/2$. Такое мы проделаем по всевозможным α и β.

Обратим внимание, что некоторые из неинтересных пар вершин были посчитаны дважды. Это такие пары вершин, что (α_x, β_x) и (α_y, β_y) совпадают точно (по обеим компонентам). Мы можем для каждой пары значений насчитать количество соответствующих вершин c и прибавить к текущему ответу $c \cdot (c-1)/2$.

В коде ниже пусть p — это массив пар номеров компонент для всех вершин, кроме a и b. Каждая пара — это номер компоненты связности этой вершины, если удалить a, и номер компоненты связности этой вершины, если удалить b. Тогда основная часть решения может выглядеть так:

```
long long ans = (long long)(p.size()) * (p.size() - 1) / 2;
{
    map<int,int> pp;
    forn(i, n - 2)
        pp[p[i].first]++;
    for (auto kv: pp)
        ans -= ((long long) kv.second * (kv.second - 1)) / 2;
}
{
    map<int,int> pp;
    forn(i, n - 2)
        pp[p[i].second]++;
    for (auto kv: pp)
        ans -= ((long long) kv.second * (kv.second - 1)) / 2;
}
map<pair<int,int>,int> pp;
forn(i, n - 2)
    pp[p[i]]++;
for (auto kv: pp)
    ans += ((long long) kv.second * (kv.second - 1)) / 2;

cout << ans << endl;
```

Для построения массива p можно воспользоваться кодом ниже:

```
void dfs(int u, int c) {
    if (color[u] == 0) {
        color[u] = c;
        for (int v: g[u])
            dfs(v, c);
    }
}

vector<int> groups(int f) {
    color = vector<int>(n);
    color[f] = -1;
    int c = 0;
    forn(i, n)
        if (i != f && color[i] == 0)
            dfs(i, ++c);
    return color;
}

// begin of read input and construct graph g
// some code ...
// end of read input and construct graph g

vector<pair<int,int>> p(n - 2);
{
    int index = 0;
    auto g = groups(a);
    forn(i, n)
        if (i != a && i != b)
            p[index++].first = g[i];
}
{
    int index = 0;
    auto g = groups(b);
    forn(i, n)
        if (i != a && i != b)
            p[index++].second = g[i];
}
```

1259F - Красивый прямоугольник

Сначала сформулируем критерий, что из заданного набора чисел $x_1, x_2, \ldots, x_k$ можно сложить красивый прямоугольник $a \times b$ (где $a \cdot b = k, a \le b$). Очевидно, что если какое-то значение встретится более a раз, то среди a строк будет существовать такая, в которой это значение встречается два или более раз (принцип Дирихле).

Покажем, что если каждое значение в $x[1 \ldots k]$ встречается не более a раз, то получится сложить красивый прямоугольник $a \times b$ (где $a \cdot b = k, a \le b$).

Занумеруем клетки от левого верхнего угла по порядку от единицы сдвигаясь каждый раз по диагонали. Пусть строки пронумерованы от 0 до $a-1$, а столбцы — от 0 до $b-1$. Начнём от клетки $(0, 0)$ и будем сдвигаться каждый раз вправо-вниз. Если дошли до края, то просто переходим по циклу. Таким образом, от клетки (i, j) мы перемещаемся каждый раз к клетке $((i+1) \bmod a, (j+1) \bmod b)$ (где $p \bmod q$ — это остаток при делении p на q). Если вдруг мы перемещаемся в уже посещенную ранее ячейку, то сделаем $i := (i+1) \bmod a$.

1	8	6
4	2	9
7	5	3

1	20	9	16	5	24
13	2	21	10	17	6
7	14	3	22	11	18
19	8	15	4	23	12

Пример нумерации для прямоугольников 3×3 и 4×6.

Можно показать, что при такой нумерации в одну строку и столбец попадают числа, которые отличаются не менее чем на $a-1$ (раз мы попали в одну строку/столбец, то сделали не менее одного оборота). При этом разница достигает значения $a-1$ (а не a), когда мы перемещаемся в ранее посещенную ячейку и делаем $i := (i+1) \bmod a$. Можно показать, что длины таких орбит равны $lcm(a, b)$ (lcm — это наименьший общий делитель), следовательно, делятся на a. Это означает, что если расставлять значения из x в порядке от наиболее часто встречающихся (в худшем случае тех, которые встречаются a раз) к наименее часто встречающимся, то всегда в каждой строке и столбце будут различные значения.

Таким образом, план решения выглядит следующим образом: найдём оптимальные a и b, что ответ является прямоугольником $a \times b$ ($a \le b$) наибольшего размера. Для этого переберем всевозможных кандидатов в a и для каждого такого кандидата каждое значение v из x можно взять в ответ не более $\min(c_v, a)$ раз, где c_v — это количество вхождений v в заданную последовательность. Таким образом, если a зафиксирован, то верхняя оценка на площадь равна сумме $\sum \min(c_v, a)$ по всевозможным различным значениям v из заданной последовательности. Следовательно, максимальное значение b это $\sum \min(c_v, a)/a$. И если $a \cdot b$ больше предыдущего найденного ответа и $a \le b$, то запомним их как новые оптимальные размеры искомого прямоугольника.

Мы можем поддерживать значение $\sum \min(c_v, a)$ при увеличении a на единицу, для этого достаточно прибавлять каждый раз к этой величине $geq[a]$, где $geq[a]$ — это количество различных значений в заданном массиве, которые встречаются a или более раз (этот массив можно предпосчитать).

Следующий код считывает входные данные и предпосчитывает массив geq.

```
int n;
scanf("%d", &n);

map<int,int> cnts;
forn(i, n) {
    int x;
    scanf("%d", &x);
    cnts[x]++;
}

vector<vector<int>> val_by_cnt(n + 1);
for (auto val_cnt: cnts)
    val_by_cnt[val_cnt.second].push_back(val_cnt.first);

vector<int> geq(n + 1);
geq[n] = val_by_cnt[n].size();
for (int i = n - 1; i >= 1; i-)
    geq[i] = geq[i + 1] + val_by_cnt[i].size();
```

Следующий код находит оптимальные размеры прямоугольника, перебирая его меньшую сторону a.

```
int tot = 0, best = 0, best_a, best_b;
for (int a = 1; a <= n; a++) {
    tot += geq[a];
    int b = tot / a;
    if (a <= b && best < a * b) {
        best = a * b;
        best_a = a;
        best_b = b;
    }
}
```

Следующий код выводит оптимальные размеры, строит искомую красивую таблицу и выводит её.

```
cout << best << endl << best_a << " " << best_b << endl;
vector<vector<int>> r(best_a, vector<int>(best_b));

int x = 0, y = 0;
for (int i = n; i >= 1; i-)
    for (auto val: val_by_cnt[i])
        forn(j, min(i, best_a)) {
            if (r[x][y] != 0)
                x = (x + 1) % best_a;
            if (r[x][y] == 0)
                r[x][y] = val;
            x = (x + 1) % best_a;
            y = (y + 1) % best_b;
        }

forn(i, best_a) {
    forn(j, best_b)
        cout << r[i][j] << " ";
    cout << endl;
}
```

Таким образом, суммарное время работы алгоритма равно $O(n \log n)$, где множитель $\log n$ берется только из-за работы с `std::map` (и от него легко можно избавиться, сделав решение линейным).

1259G - Выбывание на дереве

Сперва заметим, что количество различных итоговых последовательностей совпадает с общим количеством вариантов развития событий в описанном процессе (а именно, с количеством способов выбрать, какой жетон удалить на каждом шаге). Действительно, если мы рассмотрим любую итоговую последовательность, мы можем полностью проэмулировать процесс и определить, из какого конца ребра мы убирали жетон на каждом шаге, пропуская шаги, когда жетоны убирать не надо.

Чтобы посчитать количество вариантов, воспользуемся динамическим программированием по поддеревьям. Рассмотрим вершину v и временно забудем про все рёбра, ни один конец которых не лежит в поддереве v. Другими словами, мы оставляем только рёбра в поддереве v, а также ребро из v в её родителя p (для удобства будем считать, что из корня есть ребро с индексом n в виртуального родителя). Таким образом, мы предполагаем, что жетон из p не может быть удален никаким образом, кроме как рассматривая ребро из v.

Для краткости будем говорить, что «v сравнилась с u», подразумевая «когда мы рассмотрели ребро (v, u), в обоих концах лежал жетон», а также «v была убита u», подразумевая «v сравнилась с u и в результате потеряла жетон».

В поддереве v мы будем различать три варианта развития событий:

- v была убита до сравнения с p (ситуация 0);
- v была убита p (ситуация 1);
- v убила p (ситуация 2).

Обозначим $dp_{v,s}$ количество вариантов в поддереве v, относящихся к ситуации номер s.

Пускай $u_1, \dots, u_k$ — список детей v, упорядоченных по возрастанию $\text{index}(u_i, v)$ (индексы рёбер), и d — наибольший индекс, такой что $\text{index}(u_d, v) < \text{index}(v, p)$. Если $\text{index}(u_1, v) > \text{index}(v, p)$, положим $d = 0$.

Выпишем реккурентные соотношения для $dp_{v,s}$. Например, при $s = 0$ v была убита одним из своих детей u_i с $i \le d$. Для фиксированного i, процесс должен быть развиваться так:

- все дети $u_1, \dots, u_i$ были либо убиты до сравнения с v, либо убиты v (но не могли сравниться с v и выжить).
- v была убита u_i;
- все дети $u_{i+1}, \dots, u_k$ были либо убиты до сравнения с v, либо «выжили» несуществующее сравнение с v (но не могли быть убиты v).

Таким образом мы получаем формулу

$$dp_{v,0} = \sum_{i=1}^{d} \left(\prod_{j=1}^{i=1} (dp_{u_j,0} + dp_{u_j,1}) \times dp_{u_i,1} \times \prod_{j=i+1}^{k} (dp_{u_j,0} + dp_{u_j,2}) \right).$$

Рассуждая по аналогии, можно получить и остальные соотношение:

$$dp_{v,1} = \prod_{j=1}^{i=d} (dp_{u_j,0} + dp_{u_j,1}) \times \prod_{j=d+1}^{k} (dp_{u_j,0} + dp_{u_j,2}),$$

$$dp_{v,2} = \sum_{i=d+1}^{k} \left(\prod_{j=1}^{i=1} (dp_{u_j,0} + dp_{u_j,1}) \times dp_{u_i,1} \times \prod_{j=i+1}^{k} (dp_{u_j,0} + dp_{u_j,2}) \right) + \prod_{j=1}^{k} (dp_{u_j,0} + dp_{u_j,1}).$$

Во всех этих формулах мы естественно предполагаем, что пустые произведения равны 1.

Чтобы быстро вычислять эти соотношения, посчитаем префиксные произведения значений $dp_{j,0} + dp_{j,1}$, а также суффиксные произведения $dp_{j,0} + dp_{j,2}$. В итоге ответ равен $dp_{root,0} + dp_{root,1}$ (либо корень был убит в процессе, либо он выжил и был убит виртуальным предком).

Это решение можно реализовать за время $O(n)$, поскольку рёбра на входе уже упорядочены по индексам, но $O(n \log n)$ также должно быть достаточно.

1276E - Four Stones

First, when is the task impossible? If all a_i have the same remainder d modulo an integer g, then it will always be true regardless of our moves. The largest g we can take is equal to $\mathrm{GCD}(a_2 - a_1, a_3 - a_1, a_4 - a_1)$. *Remark: case when all a_i are equal is special.* If the GCD's or the remainders do not match for a_i and b_i, then there is no answer. For convenience, let us apply the transformation $x \to (x - d)/g$ to all coordinates, and assume that $g = 1$. We can observe further that parity of each coordinate is preserved under any operation, thus the number of even and odd numbers among a_i and b_i should also match.

Otherwise, we will divide the task into two subproblems:

1. Given four stones $a_1, \ldots, a_4$, move them into a segment $[x, x + 1]$ for an arbitrary even x.
2. Given four stones in a segment $[x, x + 1]$, shift them into a segment $[y, y + 1]$.

Suppose we can gather a_i and b_i into segments $[x, x + 1]$ and $[y, y + 1]$. Then we can solve the problem as follows:

- gather a_i into $[x, x + 1]$;
- shift the stones from $[x, x + 1]$ to $[y, y + 1]$;
- undo gathering b_i into $[y, y + 1]$ (all moves are reversible).

First subproblem. Throughout, let Δ denote the maximum distance between any two stones. Our goal is to make $\Delta = 1$. We will achieve this by repeatedly decreasing Δ by at least a quarter, then we will be done in $O(\log \Delta)$ steps.

Suppose $a_1 \le a_2 \le a_3 \le a_4$, and one of the stones a_i is in the range $[a_1 + \Delta/4, a_1 + 3\Delta/4]$. Consider two halves $[a_1, a_i]$ and $[a_i, a_4]$, and mirror all stones in the shorter half with respect to a_i. Then, a_i becomes either the leftmost or the rightmost stone, and the new maximum distance Δ' is at most $3\Delta/4$, thus reaching our goal.

What if no stones are in this range? Denote $d_i = \min(a_i - a_1, a_4 - a_i)$ — the distance from a_i to the closest extreme (leftmost or rightmost) stone. We suppose that $d_2, d_3 < \Delta/4$, otherwise we can reduce Δ as shown above. Further at least one of d_2, d_3 is non-zero, since otherwise we would have $\mathrm{GCD}(a_2 - a_1, a_3 - a_1, a_4 - a_1) = 1 = \Delta$, and the goal would be reached.

Observe that performing moves (a_i, a_j), (a_i, a_k) changes a_i to $a_i + 2(a_k - a_j)$. With this we are able to change d_i to $d_i + 2d_j$. Repeatedly do this with $d_i \le d_j$ (that is, we are adding twice the largest distance to the smallest). In $O(\log \Delta)$ steps we will have $\max(d_2, d_3) \ge \Delta/4$, allowing us to decrease Δ as shown above.

Finally, we have all $a_i \in [x, x + 1]$. To make things easier, if x is odd, move all $a_i = x$ to $x + 2$.

Second subproblem. We now have all stones in a range $[x, x + 1]$, and we want to shift them by $2d$ (we will assume that $2d \ge 0$). Observe that any arrangement of stones can be shifted by 2Δ by mirroring all stones with respect to the rightmost stone twice.

Consider an operation we will call *Expand*: when $a_1 \le a_2 \le a_3 \le a_4$, make moves (a_3, a_1) and (a_2, a_4). We can see that if we apply *Expand* k times, the largest distance Δ will grow exponentially in k. We can then shift the stones by $2d$ as follows:

1. Apply *Expand* until $\Delta > d$.
2. While $\Delta \le d$, shift all stones by 2Δ as shown above, and decrease d by Δ.
3. If $\Delta = 1$, exit.
4. Perform *Expand*$^{-1}$ — the inverse operation to *Expand*, and return to step 2.

In the end, we will have $\Delta = 1$ and $d = 0$. Further, since Δ grows exponentially in the number of *Expand*'s, for each value of Δ we will be making $O(1)$ shifts, thus the total number of operations for this method $O(\log d)$.

1276F - Asterisk Substrings

There are two types of substrings we have to count: with and without the $*$. The substrings without $*$ are just all substrings of the intial string s, which can be counted in $O(n)$ or $O(n \log n)$ using any standard suffix structure.

We now want to count the substrings containing $*$. Consider all such substrings of the form "$u*v$", where u and v are letter strings. For a fixed prefix u, how many ways are there to choose the suffix v? Consider the *right context* $rc(u)$ — the set of positions $i + |u|$ such that the suffix $s_i s_{i+1} \dots$ starts with u. Then, the number of valid v is the number of distinct prefixes of suffixes starting at positions in the set $\{i + 1 \mid i \in rc(u)\}$.

For an arbitrary set X of suffixes of s (given by their positions), how do we count the number of their distinct prefixes? If X is ordered by lexicographic comparison of suffixes, the answer is $1 + \sum_{i \in X} (|s| - i) - \sum_{i,j \text{ are adjacent in } X} |LCP(i, j)|$, where $LCP(i, j)$ is the largest common prefix length of suffixes starting at i and j. Recall that $LCP(i, j)$ queries can be answered online in $O(1)$ by constructing the suffix array of s with adjacent LCPs, and using a sparse table to answer RMQ queries. With this, we can implement X as a sorted set with lexicographic comparison. With a bit more work we can also process updates to X and maintain $\sum LCP(i, j)$ over adjacent pairs, thus always keeping the actual number of distinct prefixes.

Now to solve the actual problem. Construct the suffix tree of s in $O(n)$ or $O(n \log n)$. We will run a DFS on the suffix tree that considers all possible positions of u. When we finish processing a vertex corresponding to a string u, we want to have the structure keeping the ordered set $X(u)$ of suffixes for $rc(u)$. To do this, consider children $w_1, \dots, w_k$ of u in the suffix tree. Then, $X(u)$ can be obtained by merging $ex(X(w_1), |w_1| - |u|), \dots, ex(X(w_k), |w_k| - |u|)$, where $ex(X, l)$ is the structure obtained by prolonging all prefixes of X by l, provided that all extensions are equal substrings. Note that $ex(X, l)$ does not require reordering of suffixes in X, and simply increases the answer by l, but we need to subtract l from all starting positions in X, which can be done lazily. Using smallest-to-largest merging trick, we can always have an up-to-date $X(u)$ in $O(n \log^2 n)$ total time.

We compute the answer by summing over all u. Suppose the position of u in the suffix tree is not a vertex, but is located on an edge l characters above a vertex u. Then we need to add $l + ans(X(u))$ to the answer. Since we know $X(u)$ for all u, the total contribution of these positions can be accounted for in $O(1)$ per edge. If u is a vertex, on the step of processing u we add $ans(ex(X(w_1), |w_1| - |u| - 1) \cup \dots \cup ex(X(w_k), |w_k| - |u| - 1))$, using smallest-to-largest again. Note that we still need to return $X(u)$, thus after computing u's contribution we need to undo the merging and merge again with different shifts.

The total complexity is $O(n \log^2 n)$. If we compute LCPs in $O(\log n)$ instead of $O(1)$, we end up with $O(n \log^3 n)$, which is pushing it, but can probably pass.