Задача A. Плагиат кода

Автор задачи и разработчик: Владислав Власов

Заметим, что порядок удаления пар соседних символов не важен. При любом порядке удаления из s будут удалены несколько отрезков подряд идущих символов, каждый из которых будет четной длины. Такого же результата всегда можно добиться, удаляя пары стоящих рядом символов по очереди слева-направо.

Так же сразу заметим, что если длины s и t имеют разную четность, то ответ всегда «NO», потому что удаление двух символов сохраняет четность длины строки.

Для прохождения **первой подгруппы** достаточно написать полный перебор. Будем обрабатывать символы строки s по очереди и для каждого перебирать, удалить его вместе со следующим или нет. Решение работает за $\mathcal{O}(2^{|s|})$, что с запасом проходит по времени.

Вторая подгруппа требовала анализа строки s. Если в t есть только символы ‘a’, то все символы s, не равные ‘a’, должны быть удалены. Как уже было сказано выше, удалить можно только четные отрезки подряд идущих символов, поэтому достаточно было проверить, что при удалении всех отрезков из «лишних» символов строка t не станет слишком короткой.

Чтобы получить конечную длину строки после удаления всего лишнего, достаточно найти все отрезки «лишних символов», что можно сделать за время $\mathcal{O}(|s|)$. Пройдемся по строке s, поддерживая счетчик символов, не равных ‘a’. Когда встречается очередной символ ‘a’, очередной отрезок закончился, и надо запомнить значение счетчика, после чего обнулить его. Теперь, каждый четный отрезок можно удалить сам по себе, а каждый нечетный — только захватив стоящую рядом ‘a’.

Надо также учесть, что если строка начинается и заканчивается символом, не равным ‘a’, все отрезки нечетны, и между ними стоит по одному символу ‘a’, то в принципе невозможно все эти отрезки удалить.

В **третьей подгруппе** не требовался подробный анализ строки s. Если гарантируется, что $|s| = |t|+2$, то из s надо удалить ровно два символа. Можно перебрать позицию удаляемых символов, после чего сравнить полученную строку и t за длину строки. Такое решение работает за $\mathcal{O}(|s|^2)$.

Полное решение опирается только на идею удаления символов слева-направо. Посмотрим на первые символы s и t. Если они совпадают, то нет необходимости удалять первый символ из s. Докажем это: пусть в правильном ответе на самом деле s_1 удаляется вместе с s_2, и, возможно, следующими символами, и на первом месте стоит s_i. Тогда i — нечетное, потому что было удалено четное число символов. Но тогда на самом деле можно было удалить все символы с s_2 по s_i включительно, и строка бы не изменилась, так как $s_1 = t_1 = s_i$.

Осталось только жадно удалить все символы, которые не совпадают с соответствующим символом t. Будем идти по строке s, и, встречая $s_i \neq t_i$, будем удалять s_i вместе с s_{i+1}. Действительно, мы доказали, что если $s_i = t_i$, то его можно оставить на месте, тогда как, очевидно, если $s_i \neq t_i$, а все предыдущие символы уже совпали, s_i придется удалить.

Для полного решения надо было реализовать описанную выше идею без явного удаления символов. Удалять символы из середины строки можно только за время, пропорциональное ее длине. Чтобы получить время $\mathcal{O}(|s|)$, будем поддерживать указатель на следующий неудаленный символ s. Если он указывает на символ, равный t_i, переходим к следующим. Иначе «удаляем» два символа из s, то есть просто двигаем указатель на две позиции вперед. Если указатель дойдет до конца s раньше, чем найдется совпадение со всей t, ответ — «NO», иначе (в случае, если $|s| \bmod 2 = |t| \bmod 2$) — «YES».

Задача B. SpamGPT-4

Автор задачи: Даниил Орешников, разработчики: Константин Бац и Даниил Орешников

Для решения задачи будем анализировать каждое отправленное сообщение. Можно показать, что ответ не превышает число порядка $\frac{T^2}{2}$, поэтому в **первой подгруппе** можно было явно просимулировать все выполняемые ботом действия и обработать каждое сообщение отдельно. Такое решение работает за время $\mathcal{O}$(ответа), что помещается в ограничения по времени.

Более аккуратная симуляция процесса, работающая за время $\mathcal{O}(T)$, проходила **вторую подгруппу**. Заметим, что сообщения не обязательно отличать друг от друга, достаточно просто считать количество отправленных в какой-либо момент сообщений каждым из ботов.

Пусть $\texttt{recv}_1[t]$ и $\texttt{recv}_2[t]$ — количество сообщений, полученных первым и вторым ботом в момент времени t, соответственно. Тогда достаточно для каждого t, делящегося на a или b, увеличивать соответствующий `recv` на 1. Помимо этого, надо учесть ответы, и для каждого t увеличить $\texttt{recv}_1[t+1]$ на $\texttt{recv}_2[t]$ и наоборот. Перебрав все t от 0 до T и сложив полученные значения `recv`, получим ответ.

Две следующие подгруппы позволяют получить частичные баллы, если придумать менее общую формулу для ответа, чем в полном решении. Есть два подхода: рассмотреть каждый момент времени и посчитать, сколько сообщений каждый бот в эту секунду получил, и рассмотреть каждое отправленное не-ответное сообщение и посчитать, сколько на каждое из них было ответов.

Сразу отметим, что дальше в разборе считаются полученные сообщения, а по задаче требуется количество отправленных, но на самом деле это одно и то же: количество полученных первым ботом равно количеству отправленных вторым, и наоборот.

Для **третьей подгруппы** рассмотрим первый подход. Каждый бот отправляет по новому сообщению каждую секунду, в таком случае в момент времени t каждый бот получает ответы на все отправленные в момент времени $t-1$ сообщения и еще одно новое сообщение. Несложно заметить, что в таком случае в момент времени t каждый из ботов получает ровно $t+1$ сообщение. Осталось просуммировать $t+1$ по всем t от 0 до T. Получится сумма арифметической прогрессии, то есть

$$\text{ответ} = \frac{(T+1)(T+2)}{2}.$$

Для **четвертой подгруппы** проще рассмотреть второй подход. Поскольку $a \leqslant T \leqslant b < 2a$, первый бот успеет отправить ровно два сообщения, а второй — не больше двух (в момент времени 0, и возможно в момент времени T, если $T = b$). Для каждого отправленного сообщения легко посчитать, сколько ответов на него получит каждый из ботов. Ответы пересылаются каждую секунду, то есть любой бот получает ответ на свое сообщение каждые две секунды, начиная со следующей после момента отправки.

Таким образом, первый бот на сообщение, отправленное им в момент времени t, получит ровно $\left\lfloor \frac{T-t+1}{2} \right\rfloor$ ответов. А второй бот получит это самое сообщение или ответы на него в моменты времени t, $t+2$, $\ldots$, то есть получит всего $\left\lfloor \frac{T-t+2}{2} \right\rfloor$ сообщений, начавшихся с данного. Чтобы получить ответ, осталось просуммировать данные величины для каждого отправленного сообщения, которых не больше четырех.

Полное решение опирается на использованную выше идею. Если первый бот отправляет сообщение в момент времени ka, где $k \geqslant 0$ — некоторое целое число, то он сам получит благодаря ему $\left\lfloor \frac{T-ka+1}{2} \right\rfloor$ сообщений, а второй получит $\left\lfloor \frac{T-ka+2}{2} \right\rfloor$ сообщений. Ответ для первого бота, таким образом, равен

$$\sum_{k=0}^{\left\lfloor \frac{T}{a} \right\rfloor} \left\lfloor \frac{T-ka+1}{2} \right\rfloor + \sum_{k=0}^{\left\lfloor \frac{T}{b} \right\rfloor} \left\lfloor \frac{T-kb+2}{2} \right\rfloor,$$

а для второго — симметрично наоборот.

Посчитаем каждую сумму отдельно. Заметим, что если бы не было деления на 2, получилась бы арифметическа прогрессия с шагом a или b. Сумму арифметической прогрессии с шагом Δ, начинающейся в числе x и имеющей q элементов, можно посчитать по формуле

$$qx + \Delta\frac{q(q-1)}{2}.$$

Если представить выражение $\left\lfloor \frac{r_1}{2} + \frac{r_2}{2} + \ldots + \frac{r_q}{2} \right\rfloor$ как $\frac{r_1+r_2+\ldots+r_q}{2} - \frac{(r_1 \bmod 2)+(r_2 \bmod 2)+\ldots+(r_q \bmod 2)}{2}$, то остается только посчитать сумму остатков членов данной прогрессии по модулю 2 и вычесть, после чего поделить результат на 2.

Для этого достаточно посчитать, сколько членов прогрессии имеют остаток 1 по модулю 2. Если мы считаем прогрессию из элементов вида $T-ka+1$, то в зависимости от четности T и a можно за

$\mathcal{O}(1)$ посчитать количество нечетных элементов. Соответственно, теперь мы умеем вычислять сумму прогрессии и сумму остатков ее элементов по модулю 2, а значит можем и вычислить исходную сумму за время $\mathcal{O}(1)$.

Задача C. Есть n стульев...

Автор задачи и разработчик: Владислав Власов

Заметим, что любой набор выгодно рассматривать в отсортированном по высоте порядке, иначе разница высот некоторой пары соседних стульев будет больше, чем если бы стулья были отсортированы. Отсортируем стулья по высоте, теперь нас интересует некоторая подпоследовательность полученного массива.

Заметим, что на самом деле нас интересует не просто подпоследовательность, а отрезок массива. Если бы оптимальным ответом была подпоследовательность не идущих подряд стульев, можно было бы добавить все «пропущенные», и суммарная ширина бы увеличилась, а максимальная разность высот между соседними — уменьшилась бы.

Для решения **первой подгруппы** достаточно перебирать все возможные отрезки и проверять каждый за линейное время. Такое решение работает за $\mathcal{O}(n^3)$.

Решение **второй подгруппы** похоже на решение первой, но для этой подгруппы нужно быстро обрабатывать каждый отрезок. Чтобы находить суммарную ширину стульев на отрезке, можно воспользоваться префиксными суммами, а чтобы находить максимальную разность соседних по высоте — либо реализовать операцию «минимум на отрезке» с поддерживающих ее структур данных, либо просто перебирать отрезки по левой границе, а при фиксированной левой границе — по возрастанию правой, одновременно обновляя максимум при расширении отрезка. Время работы такого решения — $\mathcal{O}(n^2)$.

В **третьей подгруппе** нам достаточно проверять отрезки длины ровно H, поскольку именно столько необходимо, чтобы Влад поместился, а расширение отрезка не может уменьшить его неудобность. Таким образом решение сводится к задаче «максимум в окне», которую можно решать за $\mathcal{O}(n \log n)$ с помощью `std::set` или же за $\mathcal{O}(n)$ с помощью очереди с поддержкой минимума.

В **четвертой подгруппе** ответ не превосходит 30. Тогда можно перебрать ответ и проверить, можно ли получить ответ не хуже такого, за $\mathcal{O}(n)$. Для проверки будем поддерживать текущий отрезок и идти слева-направо, жадно набирая элементы. Как только какой-то элемент не получается добавить из-за большой разности с предыдущим, начнем новый отрезок с него.

Есть несколько вариантов **полного решения**. Можно воспользоваться идеей четвертой подгруппы, заменив перебор ответа бинарным поиском по ответу. Альтернативно можно воспользоваться идеей второй и третьей подгрупп, но сократить перебор отрезков до $\mathcal{O}(n)$ с помощью техники двух указателей, поддерживая для фиксированной левой границы ближайшую правую, суммарная ширина стульев между которыми достаточна. Максимальную разность на отрезке можно поддерживать тем же `std::set`.

Задача D. Перекладывание ответственности

Автор задачи: Даниил Орешников, разработчик: Мария Жогова

Первая подгруппа расчитана на прямую реализацию описанного в условии процесса. Будем поддерживать указатель на текущую задачу x и количество уже распределенных в каждой задаче элементов $\texttt{cnt}[i]$. Тогда, выдавая разработчику очередной элемент, возьмем очередной элемент из задачи x, увеличим $\texttt{cnt}[x]$ на 1, после чего будем перемещать x по кругу вперед, пока не встретим следующую задачу, у которой $\texttt{cnt}[i] < c_i$. Решение работает за время $\mathcal{O}(n \cdot \max(c_i))$, поскольку перемещений x будет не больше, чем столько.

Во **второй подгруппе** можно было легко набрать баллы, проанализировав сколько элементов какой из двух задач достанется каждому разработчику. Не теряя общности, пусть $c_1 \leqslant c_2$, тогда первые $2c_1$ элементов будут распределяться по очереди из двух задач, а оставшиеся будут взяты из второй. В таком случае первому разработчику достанутся элементы суммарной сложности $\left\lceil \frac{c_1}{2} \right\rceil \cdot w_1 + \left\lfloor \frac{c_1}{2} \right\rfloor \cdot w_2$, а второму — все оставшиеся. Случай $c_1 > c_2$ полностью аналогичен.

Для **третьей подгруппы** также работает решение первой, однако приведем более сложное решение, которое будет шагом на пути к полному. Введем определение «фазы», которое будет дальше полезно для полного решения. Фазой t назовем отрезок времени, в который распределяются t-е элементы из каждой задачи, в которой есть хотя бы t элементов. Фазы, таким образом, будут иметь номера от 1 до $\max c_i$.

В условиях третьей подгруппы фаз будет не больше двух. Если все $c_i \leqslant 2$, то каждый разработчик, кроме, возможно, одного, получит один или два своих элемента в рамках одной фазы. Для того разработчика, которому, возможно, достанется последний элемент из первой фазы и первый из второй, посчитаем ответ отдельно, а для остальных достаточно построить префиксные суммы на участвующих в каждой фазе задачах, и находить итоговый вес как сумму на отрезке в рамках одной фазы.

Для следующих подгрупп обобщим описанное решение. Заметим, что среди всех фаз есть не более n различных. Действительно, все фазы с первой по $\min c_i$ одинаковы и состоят из всех задач по очереди. Следующие фазы до второго по величине c_i одинаковы и состоят из всех задач, кроме тех, в которых $c = \min c_i$, и так далее.

Заметим, что элементы, достающиеся очередному разработчику, образуют либо отрезок какой-то фазы, либо суффикс последней фазы, некоторое целое число следующих фаз целиком (возможно, ноль) и какой-то, возможно пустой, префикс следующей фазы. Отсортируем все задачи по убыванию c_i и посчитаем префиксные суммы w_i на полученном порядке задач.

Теперь будем перебирать разработчиков по очереди, поддерживая номер фазы и номер задачи в фазе, на которых мы остановились на предыдущем разработчике. Для решения **четвертой подгруппы** достаточно найти сумму на соответствующем отрезке или суффиксе текущей фазы за $\mathcal{O}(n)$, после чего обработать все предназначенные ему целиком фазы, и снова за $\mathcal{O}(n)$ добавить элементы из префикса следующей фазы, если надо.

Чтобы быстро обработать фазы, которые целиком отойдут данному разработчику, посмотрим на номер фазы p, в начале которой мы сейчас находимся: пусть он лежит между c_i и c_{i+1}, где c отсортированы по возрастанию. Тогда следующие $c_{i+1} - p + 1$ фаз одинаковы, и в каждой из них выбираются элементы из $n-i+1$ оставшихся задач. Сумму w_i всех задач в фазе мы предподсчитали, поэтому, если разработчику осталось выдать $\texttt{rem} \leqslant (c_{i+1} - p + 1) \cdot (n - i)$ элементов, выдадим ему $\left\lfloor \frac{\texttt{rem}}{n-i} \right\rfloor$ следующих фаз целиком за $\mathcal{O}(1)$, и префикс следующей за ними фазы размера $\texttt{rem} \bmod (n-i)$. Иначе, выдадим ему все оставшиеся фазы этого «блока» и перейдем к рассмотрению фаз между c_{i+1} и c_{i+2}.

Действия, которые выполняются «долго» — взятие отрезка/суффикса и префикса фазы. Помимо этого, для одного разработчика могут быть рассмотрены несколько «блоков» фаз между соседними c_i и c_{i+1}. Однако, вычисление суффикса и префикса производится для каждого разработчика не больше одного раза, а перемещений к следующему «блоку» одинаковых фаз будет не больше n. Итого, суммарно мы потратили $\mathcal{O}(n)$ времени на обработку всех фаз, не считая вычисления суммы на отрезках, суффиксах и префиксах, а на них — $\mathcal{O}(n^2)$ времени, чего достаточно для прохождения четвертой подгруппы.

Для пятой и шестой подгруппы требовалось соптимизировать нахождение сумм на отрезках, суффиксах и префиксах внутри фаз. Сразу сведем эту задачу к поиску суммы на префиксе с помощью префиксных сумм. Осталось научиться быстро находить сумму на префиксе в фазе быстрее, чем за $\mathcal{O}(n)$.

В **пятой подгруппе** подходил любой способ находить сумму за $\mathcal{O}(\log^2 n)$. Мы не будем на них фокусироваться, так как эти способы достигаются идеями, очень похожими на идеи полного решения, но в каких-то местах менее оптимальными. Для **полного решения** можно было воспользоваться, например, одной из двух следующих идей.

Первая идея основана на дереве отрезков. Будем хранить дерево отрезков, в котором i-й элемент равен w_i, если в i-й задаче еще остались нераспределенные элементы, и 0 иначе. При переходе между фазами достаточно просто обнулить «закончившиеся» задачи. Суммарно это займет $\mathcal{O}(n \log n)$ времени как n запросов к ДО. Чтобы получить сумму первых k задач в фазе, то есть сумму на префиксе, надо сначала получить позицию k-го ненулевого элемента t, а затем сделать запрос суммы на

отрезке $[0, t]$. Чтобы находить t, надо было хранить количество ненулевых элементов в каждом отрезке ДО, и либо делать бинпоиск и запросы количества нулей на отрезке за $\mathcal{O}(\log^2 n)$ (чего хватало для пятой подгруппы), либо делая спуск по дереву за $\mathcal{O}(\log n)$.

Альтернативное решение использовало декартово дерево по ключу, равному номеру задачи. Будем хранить в вершинах ДД w_i и поддерживать их сумму на поддеревьях. Аналогично, переход между фазами — это удаление нескольких элементов из ДД по ключу, а запрос суммы на префиксе фазы заключается в поиске k-й порядковой статистики, и взятия суммы в левом поддереве после `split` по найденному ключу. Обе операции можно выполнять за $\mathcal{O}(\log n)$, если поддерживать также размеры поддеревьев.

Задача E. Быстрый исполнитель

Автор задачи и разработчик: Даниил Орешников

Первая подгруппа, стандартно, позволяет набрать баллы, написав реализацию описанного в условии процесса. Присвоим $b \leftarrow a$, и честно выполним все $m \cdot p$ операций. Каждая операция выполняется для каждого элемента, поэтому время работы получается $\mathcal{O}(nmp)$.

Во **второй подгруппе** $m = 1$, что, в частности, означает, что всегда применяется одна и та же операция. Сразу приведем решение второй и **третьей подгруппы** вместе, потому что они почти ничем не отличаются. Посмотрим на каждый элемент последовательности отдельно. Всего применяется $m \cdot p$ операций, при каждой операции a сдвигается на d влево. Таким образом, b_i можно вычислить как

$$b_i = a_i \circ a_{(i+d) \bmod n} \circ a_{(i+2d) \bmod n} \circ \ldots \circ a_{(i+mpd) \bmod n},$$

где за $\circ$ обозначена применяемая операция.

Заметим, что для каждого элемента массовое применение такой операции можно вычислять быстро. А именно, разобьем весь a на «циклы» по сдвигу на d. Таких циклов будет ровно $\texttt{gcd}(n, d)$, и для каждого i указанная выше последовательность элементов a будет образовывать некоторый отрезок на цикле (возможно, с несколькими повторениями цикла целиком). Для операций `and` и `or` можно заметить, что целиком вошедший в последовательность много раз цикл можно отбросить, и оставить только «хвост» не больше цикла. Для операции `xor` надо учесть лишний раз `xor` всех элементов цикла, если цикл вошел в последовательность нечетное число раз.

Теперь, чтобы для каждого i посчитать значение операции на отрезке фиксированной длины, достаточно пройтись по всем циклам «окном» фиксированной ширины, поддерживая определенную информацию. Для `xor` это будет просто `xor` всех элементов на окне, для `or` и `and` — наличие в окне в каждом разряде единицы или нуля, соответственно. Время работы такого решения равно $\mathcal{O}(n \log \max(a_i))$.

Решения следующих подгрупп являются модификациями описанного решения. Будем рассматривать каждую из m операций отдельно, тогда для нее последовательность элементов, образующих в конечном итоге b_i, образует свой цикл, уже с шагом md вместо d.

В **четвертой подгруппе** нет операций `xor`, поэтому достаточно в каждом бите независимо найти последнюю операцию вида $b_i \leftarrow b_i$ `or` 1 или $b_i \leftarrow b_i$ `and` 0, которые задают фиксированное новое значение b_i вне зависимости от старого. Остальные операции b_i не меняют. Для каждой из m операций найдем соответствующее ей последнее имеющее значение применение этой операции в цикле. Опять же, для этого можно воспользоваться «скользящим окном» фиксированной ширины, либо же за $\mathcal{O}(n)$ найти для каждого элемента позицию ближайшего нуля и ближайшей единицы в каждом разряде.

Пятая подгруппа гарантирует, что достаточно обработать только один бит в каждом числе. Это позволяет неэффективным решениям с той же идеей набрать баллы за счет того, что надо обработать в ≈ 30 раз меньше независимых бит в каждом числе.

Для **полного решения** так же требовалось находить `xor` на отрезках в цикле. Однако надо было заметить, что нас интересует только множество операций `xor` после последнего действующего на значение `and` или `or`. Все до этой последней действующей операции не имеет значения, а после нее имеют значение только `xor`'ы. Чтобы находить `xor` на произвольных отрезках в цикле, достаточно

посчитать аналог префиксных сумм и вычислять его как `xor` двух префиксов. Время работы всего решения получается $\mathcal{O}(nm \log \max(a_i))$.

Есть альтернативное полное решение, работающее за время $\mathcal{O}(n \log \max(a_i) + n \log p)$. Можно обрабатывать все биты одновременно, преобразовав применяемые к числам операции.

Заметим, что относительно каждого бита любая композиция трех данных операций с заданными вторыми аргументами — просто некоторая булева функция от изначального первого аргумента. Воспользуемся идеей двоичного подъема и найдем для каждого i функцию $\mathtt{op}[q][i]$, соответствующую применению 2^q блоков по m операций, используя в качестве вторых аргументов элементы с шагом d, начиная с i-го. Переход от q к $q+1$ осуществляется как вычисление композиции левой и правой половины операций.

Если теперь аналогичным образом посчитать композиции не только 2^q блоков, а $p \gg q$ блоков по всем целым неотрицательным q, останется только для каждого бита исходного числа за $\mathcal{O}(1)$ применить полученную операцию.

Задача F. Устный счет

Автор задачи и разработчик: Рябчун Владимир

Будем называть блоком одно слагаемое (которое может состоять из нескольких множителей) в данном выражении. За L будем обозначать максимальную длину числа, C будет равно 10 (количество различных цифр).

Для решения **первой подзадачи** достаточно было аккуратно перебрать все пары позиций, где будет заменена цифра, и вычислить значение полученного выражения за линейное время. Асимптотика такого решения составляет $\mathcal{O}(L^2C^2n^3)$, что при достаточной аккуратности реализации укладывается в ограничения по времени.

Решение **второй подзадачи** требовало предпосчитать префиксные и суффиксные суммы значений блоков и префиксные и суффксные произведения внутри одного блока. Тогда, при изменении цифр в числах из разных блоков или изменении цифр в одном числе, данная информация позволяла вычислить значение выражения за $\mathcal{O}(1)$. В случае изменения цифр в двух различных числах внутри одного блока, для вычисления произведения чисел между ними предлагалось либо просто предпосчитать их, либо использовать алгоритм деления по модулю (который, однако, требовал дополнительного рассмотрения нулей). Итоговое решение с предпосчетом имеет асимптотику $\mathcal{O}(n^2 + n^2L^2C^2)$.

Для решения всех следующих подзадач выделим три принципиально различных случая:

1. одна или две цифры изменены в одном числе;
2. две цифры изменены в числах из разных блоков;
3. две цифры изменены в различных числах из одного блока.

Разбор первого случая можно реализовать таким способом: переберем число и вычислим, каким оно должно быть, чтобы получилось равенство. Для этого все блоки без этого числа перенесем в другую часть равенства, а потом поделим на оставшиеся в этом блоке. Если среди оставшихся есть 0, то результат зависит от выражения справа (либо равенство уже есть, либо оно невозможно). Иначе сравним исходное число и проверим, правда ли что оно и результат отличаются не более чем в двух позициях. Это решение будет работать за $\mathcal{O}(n \log \mathrm{MOD})$ с вычислением обратного по модулю. В группах 4 – 6 этот случай рассматриваться не будет.

В **третьей подзадаче** ввиду малых ограничений на значения чисел, было возможно для каждого блока явно хранить количество цифр от 0 до 9. Более того, возможно за $\mathcal{O}(nC)$ предпосчитать всевозможные степени чисел от 1 до 9.

Для случаев 1 и 3 можно в каждом блоке перебрать пару цифр — старую и новую. Пользуясь предпосчитанными значениями для фиксированного изменения, проверить ответ за $\mathcal{O}(C)$, что даст итоговую асимптотику $\mathcal{O}(nC^3)$, хотя это можно сделать и несколько быстрее.

Случай 2 также не вызывает трудностей — для каждого блока будем помнить возможные изменения значения при замене одной цифры на другую. Для одного блока таких чисел будет не более C^2. Пройдем по блокам слева направо, в каждом блоке переберем изменение и проверим, если ли нужное второе изменение среди запомненных. Это будет работать за $\mathcal{O}(nC)$.

В **четвертой подзадаче** невозможен случай 3. Разберем случай 2. Для каждого числа переберём все возможные замены одной цифры на другую, для каждой такой замены вычислим, насколько изменится значение этого числа. Будем идти по сумме слева направо и перебирать изменение в данном числе. Тогда мы знаем, какое изменение должно внести второе число. Будем поддерживать множество возможных изменений значений и искать в нем нужное число и изменение. Сложность такого решения $\mathcal{O}(nLC)$.

Пятая подзадача гарантирует отсутствие случая 2. Опишем здесь способ решения, который будет также применяться в подзадачах 6 и 7.

Предположим, что были изменены числа x и y на d_1 и d_2 соответственно. Пусть P — произведение блока, а R — требуемое значение. Тогда необходимо, чтобы $(x+d_1)(y+d_2)\frac{P}{xy}$ было равно R. Случай нулевых x или y требует отдельного рассмотрения, но он не очень сложен — деление здесь на самом деле представляет собой исключение множителя из произведения, нужно аккуратно рассмотреть случаи одного, двух и более нулей в блоке.

Далее будем предполагать, что в блоке нет нулей. Тогда выражение можно преобразовать к виду $\left(1+\frac{d_1}{x}+\frac{d_2}{y}+\frac{d_1}{x}\frac{d_2}{y}\right)=\frac{R}{P}$.

Но здесь x и y вместе с их изменениями никак не связанны, поэтому можно перебрать y и d_2 и проверить, встречали ли мы раньше необходимые x и d_1.

Данное решение будет работать за $\mathcal{O}(nLC)$.

Решения двух последних подгрупп представляют собой объединение прошлых решений. Для получения полного балла требуется аккуратная реализация — предпосчитать как можно больше делений по модулю, чтобы не вычислять их несколько раз, использовать `unordered_map` или более быстрые ассоциативные контейнеры и оптимизировать программу по объему используемой памяти и количеству реаллокаций.