День 2

Задача WeAreTheChampions. Поеду домой

Докажем, что если существуют хотя бы четыре различных пары индексов с общей суммой ($a_{x_1} + a_{y_1} = a_{x_2} + a_{y_2} = \ldots = a_{x_4} + a_{y_4}$), то обязательно найдутся две такие пары индексов, что все четыре индекса в них различны.

Разберём случаи:

- Пусть имеются 4 пары вида $(x, y_1), (x, y_2), (x, y_3), (x, y_4)$ с суммой S. Тогда $a_x + a_{y_1} = a_x + a_{y_2} = a_x + a_{y_3} = a_x + a_{y_4} = S$, из чего можно сделать вывод, что $a_{y_1} = a_{y_2} = a_{y_3} = a_{y_4}$, а значит, в качестве ответа подходят пары (y_1, y_2) и (y_3, y_4).
- Теперь пусть имеются 3 пары вида $(x, y_1), (x, y_2), (x, y_3)$, а в четвертой индекс x не фигурирует. Тогда какой бы ни была четвертая пара (z, w), в ней обязательно не будет фигурировать индекс x, а также хотя бы один из индексов y_1, y_2, y_3, а значит, мы сможем взять в качестве ответа эту пару (z, w) и какую-то из тех трех, в которых фигурирует x.
- Аналогичным образом разбираются и другие случаи. Чтобы убедиться, что ответ всегда найдется, можно представить граф, в котором вершинами являются индексы, и существует ребро между вершинами, если соответствующая пара индексов имеет сумму S. Если в таком графе есть хотя бы четыре ребра и при этом степень всех вершин не больше двух (б'ольшую степень мы исключили, разобрав предыдущие случаи), то всегда найдутся два с непересекающимися концами.

Как, используя это, найти ответ? Запустим простой перебор за $\mathcal{O}(n^2)$, который для каждой суммы будет сохранять все найденные пары с такой суммой, и для каждой очередной пары проверять, нет ли непересекающейся с ней по индексам пары среди уже найденных.

Заметим, что это работает за $\mathcal{O}(min(n^2, n + C)$, потому что как только мы обнаружим какую-то в сумму в четвёртый раз, мы немедленно вернём ответ. А если такой нет, то мы сделали не более $\mathcal{O}(C)$ итераций перебора.

Задача ThisIsGonnaHurt. Вставить текст

В первой группе можно написать самое простое и прямое решение — перебрать все 2^m разбиения, для каждого разбиения перебрать блоки и проверить, что внутри каждого блока найдётся нужная пара подстрок. Проверить каждый блок можно прямым перебором всех пар строк, проверяя на равенство проходом по строке. Итоговая сложность по времени $O(2^m \cdot m^2 k^2)$.

Для решения всех следующих групп нужно понять, как избавиться от прямого перебора всех блоков. Делается это несложно: воспользуемся техникой динамического программирования. Обозначим за $d[i]$ минимальное число блоков в разбиении, если обрезать все тексты справа до длины не больше i. Иными словами, для всех строк, длина которых больше i, мы рассматриваем только первые i символов, и ищем разбиение таких строк для $m = i$. Тогда мы можем перебрать только самый правый блок разбиения $[j; i]$, а оставшийся префикс $[1; j - 1]$ разбить на $d[j - 1]$ блоков. Таким образом, для каждого $j \leqslant i$ такого, что блок $[j; i]$ интересный, мы пытаемся улучшить текущий ответ $d[i]$ числом $d[j-1]+1$. Чтобы восстановить искомое разбиение, вместе с минимальным числом

блоков $d[i]$ мы можем хранить также $p[i]$ — указатель на левую границу последнего блока разбиения. Тогда если для некоторого интересного блока $[j;i]$ значение $d[j-1]+1 < d[i]$, то мы обновляем $p[i] := j$.

Во второй группе можно написать такое же решение, как и в первой, но прямой перебор всех разбиений заменить на динамику. Таким образом, всего m точек, в каждой за $O(m)$ перебираем левую границу блока, а каждый блок проверяем за $O(k^2m)$. Итого, получили решение за $O(m^3k^2)$ времени, которое проходит первые две группы.

В третьей группе мы всё ещё можем перебирать все левые границы для каждой правой, но проверку блока нужно оптимизировать. Для этого воспользуемся методом полиномиального хеширования. Посчитаем хеши всех префиксов прямых и перевёрнутых версий всех строк. Тогда для фиксированного блока мы можем за $O(k)$ получить хеши соответствующих подстрок всех строк и их перевёрнутых версий. Теперь, построив на хешах прямых подстрок структуру-аналог `std::set` или `std::unordered_set` (для нас важно, чтобы это была структура-множество, позволяющая для набора объектов быстро отвечать на запрос "есть ли объект X в наборе"), для каждой перевёрнутой подстроки мы можем проверить, есть ли строка с таким же хешом в наборе. Итого, проверка блока занимает $O(m^2k)$ или $O(m^2k\log k)$, в зависимости от реализации — оба варианта проходят тесты группы.

Предполагаемые решения в шестой и четвёртой группах основаны на одной идее, но различаются в проверке каждого блока. Общая идея — можем заметить, что если для некоторого c блок $[c-d;c+d]$ интересный для некоторого d, то блок $[c-d+1;c+d-1]$ также интересный. Тогда и для любого $d' \leqslant d$ блок $[c-d';c+d']$ интересный. Аналогично, для блоков чётной длины, если $[c-d;c+1+d]$ — интересный блок, то и $[c-d+1;c+d]$ — интересный блок, значит, для любого $d' \leqslant d$ $[c-d';c+1+d']$ — интересный блок. Таким образом, для фиксированного c мы можем двоичным поиском подобрать максимальное d_1 такое, что $[c-d_1;c+d_1]$ — интересный блок, и максимальное d_2 такое, что $[c-d_2;c+d_2+1]$ — интересный блок. Затем, можем насчитать значения динамики, перебирая для фиксированного i не левую границу блока, а его центр (то самое значение c), и проверяя, что d_1 (или d_2 для чётной длины) не меньше $i-j$ (или $i-j+1$). Решение, использующее такую идею, работает за $O(mT\log m + m^2)$, где T — время на проверку блока.

В шестой группе $k=1$, поэтому чтобы проверить блок, нам нужно просто проверить, что текущая подстрока является палиндромом. Для этого воспользуемся полиномиальным хешированием, как в третьей группе. Итого решение работает за $O(m\log m + m^2) = O(m^2)$, но так как за $O(m^2)$ работает вторая часть, а в ней выполняются очень простые операции, по времени такое решение проходит.

В четвёртой группе для проверки блока можно воспользоваться тем же методом с хеш-таблицей (`std::unordered_map` и аналоги), что и в третьей группе. Тогда время работы решения $O(mk\log m + m^2)$, что вписывается в ограничения. Стоит заметить, что в третьей и четвертой группах одинарные хеши могут не пройти все тесты, так как очень велика вероятность коллизий (см. парадокс дней рождений). Можно реализовать хеширование с двумя (или больше) основаниями или модулями. Чтобы решение очень трудно было взломать (подобрать контртест), лучше всего зафиксировать один модуль и хранить несколько хешей со случайными основаниями.

В пятой группе гарантируется, что существует разбиение на один или два блока. Тогда можем для каждого префикса $[1;i]$ и суффикса $[i;m]$ вычислить индикаторы $p[i]$ и $s[i]$ — индикатор равен 1, если соответствующий блок интересный, и 0 в противном случае. Насчитать такие массивы можно легко за $O(L)$, а затем перебрать границу раздела блоков или проверить, что разбиение на один блок интересно. Итого решение работает за $O(L)$.

Решения для седьмой и восьмой групп существенно используют дерево палиндромов и некоторые факты о нём. Расписывать их здесь не имеет смысла, так как это уже было сделано ранее, прочитать можно здесь: https://codeforces.com/blog/entry/56601 (разбор задачи E).

Решения для девятой группы представляют собой неоптимальные реализации полного, поэтому сразу разберём полное решение.

Мы хотим научиться быстро пересчитывать значение $d[i]$, то есть быстро искать среди всех подходящих j такое, что $d[j-1]$ минимально. Построим общее суффиксное дерево для всех строк набора и их перевёрнутых версий. Переберём первую строку пары S_l. Обозначим для простоты $S = S_l$, то-

гда $S[l;r] = S_l \dots S_r$, если $l \leqslant r$, и $S[r;l] = S_r \dots S_l$, если $r \geqslant l$. Переберём вершину x суффиксного дерева, соответствующую подстроке $S[i;1]$ (т.е некоторый суффикс перевёрнутой S). Тогда любой предок A вершины x в суффиксном дереве соответствует некоторому префиксу этой подстроки $S[i;p]$ ($p \leqslant i$). Заметим, что если для некоторой строки исходного набора S_r существует суффикс $S_r[j;|S_r|]$ такой, что $i-|A|+1 \leqslant j \leqslant i$ и $S[i;p]$ — префикс $S_r[j;|S_r|]$, то блок $[j;i]$ является интересным (потому что $S[p;i]$ — перевёрнутая строка вершины A, значит, $S[j;i]$ — перевёрнутый префикс строки A, но $S_r[j;i]$ — также прямой префикс строки A, ведь строка A это префикс $S_r[j;|S_r|]$). Можем видеть, что любой такой суффикс лежит в поддереве вершины A. Тогда при фиксированном i мы можем рассмотреть все суффиксы $S_r[j;|S_r|]$ ($j \leqslant i$) строк S_r, чья длина не меньше i. Для каждого такого суффикса рассмотрим вершину в суффиксном дереве, которая соответствует строке $S_r[j;i]$. Тогда если она является предком вершины x, то блок $[j;i]$ интересный. Если мы будем поддерживать каким-то образом некоторую информацию о положении в дереве таких подстрок $S_r[j;i]$ для всех r так, что сможем быстро искать значение j с минимальным $d[j-1]$ среди всех подстрок $S_r[j;i]$, которым соответствует вершина-предок x, то мы решим задачу. Для этого воспользуемся heavy-light декомпозицией дерева. Для каждого суффикса $S_r[j;S_r]$ мы проделаем путь от корня до вершины, соответствующей $S_r[j;S_r]$. Такой путь разбивается на $O(\log L)$ путей HLD. Тогда для каждого суффикса мы предподсчитаем, при каких i и на какой путь мы перейдём с текущего — всего $O(L \log L)$ информации, и будем поддерживать в процессе насчитывания значений динамики для каждого пути, какие суффиксы в нём лежат (и значения $d[j-1]$ для них). Путь от x до корня разбивается также на $O(\log L)$ префиксов путей HLD, поэтому мы хотим искать индексы с минимальными $d[j-1]$ на префиксах таких путей. Заметим, что нам не обязательно поддерживать для каждого $[j;i]$, какая именно вершина дерева ему соответствует — достаточно лишь хранить в нужном пути само j и значение $dp[j-1]$. Тогда для каждого пути HLD нам нужно найти среди всех $j \leqslant q$ для некоторого q индекс j_0 с минимальным $dp[j_0-1]$, и нужно поддерживать быстрое добавление и удаление пары $(j, dp[j-1])$ для пути HLD. Подходящей структурой данных является декартовое дерево по явному ключу — оно поддерживает вставку, удаление из произвольного места, разбиение по значению q на две части и поиск минимума в дереве, и всё за $O(\log L)$. Итого, если каждому суффиксу $S_r[j;|S_r]$ исходной строки сопоставить собственное декартово дерево из одной вершины, а затем вставлять их в нужные пути и удалять, то мы сможем решить задачу за $O(L \log^2 L)$, что нас устраивает.

Задача TheNextEpisode. Плитка для ванной

Обозначим за $ans_{i,\,j}$ размер максимального подквадрата, который Костя сможет купить, если его левый верхний угол будет находиться в точке $(i,\,j)$. Понятно, что любой подквадрат со стороной, меньшей, чем $ans_{i,\,j}$ также будет доступен Косте. Тогда если для каждого возможного значения запомнить сколько есть (i,j) с таким $ans_{i,j}$, то ответ после этого можно найти с помощью частичных сумм.

Теперь надо понять, как можно посчитать $ans_{i,\,j}$. Можно перебирать по возрастанию длину квадрата, после чего проверять, что он подходит, перебирая все элементы в нем. Такое решение работает за $O(n^5)$.

Первой оптимизацией будет последовательное увеличение квадрата. А именно, с каждым увеличением стороны, добавляется всего $O(n)$ клеток, и нет необходимости пересчитывать все остальные. Если поддерживать глобальный массив подсчета, где в ячейке для цвета хранить его частоту, то можно получить решение за $O(n^4)$.

Далее заметим свойство нашего ответа: $ans_{i,\,j} \geqslant ans_{i,\,j-1} - 1$. Это верно, потому что квадрат $S(i,j,k-1)$ является подквадратом $S(i,j-1,k)$ для любого k. Значит, что можно перебирать размер стороны не от единицы, а начиная с $ans_{i,j-1}$. Воспользуемся идеей похожей на префикс-функцию: в рамках движения по рамках одной горизонтальной полосы будет не более n падений размера квадрата, а значит и не более $\mathcal{O}(n)$ увеличений. Таким образом, получаем решение за $O(n^3)$.

Как перейти к более быстрому решению? Воспользуемся тем, что цветов в квадрате очень мало. Давайте для каждой точки посчитаем ближайшие $q+1$ различных цветов справа в массиве входных данных $colors_{i,\,j}$. Как это сделать? Заведём для каждого кусочка плитки массив на $q+1$ элементов, в котором будем хранить цвет плиток и самую левую плитку такого цвета в нашей полосе. Движемся

по полосе справа налево и заполняем массив для текущей ячейки используя значение $colors_{i,j}$ и массив для предыдущей просмотренной ячейки. Это можно сделать за $O(q)$ так, чтобы цвета в $colors_{i,j}$ всегда были отсортированы по возрастанию левой границы.

Получается, что теперь для каждой полосы, начинающейся в точке (i, j), мы понимаем, какое ограничение на правую границу даст нам не более, чем q цветов. Это будет самое левое вхождение $q+1$-го цвета. Осталось придумать, как объединять такие полоски в прямоугольники.

Для объединения соседних полос достаточно выполнить *merge* двух отсортированных списков, чтобы получить $q+1$ ближайший цвет для объединения двух полос. Реализовать такое слияние можно за $O(q)$.

Получается, что объединив k полос, мы можем узнать, разрешен ли квадрат $k \times k$ — надо просто проверить, что первое вхождение $q+1$-го цвета происходит хотя бы на k вертикалях правее текущей позиции. Для объединения полос можно воспользоваться очередью на двух стеках. Будем двигаться сверху вниз, пользуясь неравенством $ans_{i+1,j} \geqslant ans_{i,j} - 1$. Тогда при сдвиге надо удалить верхнюю полосу из очереди, после чего добавлять какое-то количество полос снизу. Получившееся решение имеет асимптотику $O(n^2 q)$.

Задача WindOfChange. Фокус с подмножествами

Напомним, что мы называем число x *удачным*, если исходя из размера некоторого подмножества S можно однозначно сделать вывод о том, что верно ли, что сумма элементов этого подмножества не превосходит x. Если число x не является удачным, то мы называем его *неудачным*.

Математическим языком можно записать, что x — неудачное $\Leftrightarrow \exists\, S_1, S_2 \subset S$, такие что $|S_1| = |S_2|$, $sum(S_1) \leqslant x$, $sum(S_2) > x$.

Для того, чтобы решить первую подзадачу, для каждого размера $1 \leqslant k \leqslant |S|$ переберем все возможные подмножества размера k. Для каждого из них посчитаем сумму, среди этих сумм найдем s_{min}, s_{max} — минимальную и максимальную из этих сумм. Тогда все числа полуинтервала $[s_{min}, s_{max})$ будут неудачными. Таким образом мы получим $|S|$ полуинтервалов, таких что множество всех неудачных чисел это объединение всех полуинтервалов. Тогда для того, чтобы найти ответ на задачу найдем длину объединения этих полуинтервалов. Получим решение за $O(q|S|2^{|S|})$.

Во второй подзадаче у нас есть ограничение на то, что все элементы множества S не превосходят $c = 100$. Тогда $|S| \leqslant c$ и $sum(S) \leqslant c^2$. Поскольку все неудачные числа $< sum(S)$, переберем все возможные числа x от 1 до $sum(S) - 1$. Зная число x мы можем найти максимальный размер подмножества, сумма элементов которого $\leqslant x$ и минимальный размер подмножества, сумма элементов которого $> x$. Это можно сделать простым линейным жадным алгоритмом, если у нас есть элементы множества S в отсортированном порядке. Например, если мы ищем максимальный размер подмножества, сумма элементов которого не превосходит x, то будем идти в порядке возрастания по элементам S и набирать их в подмножество до тех пор, пока сумма не превышает x. Заметим, что в итоге если мы получим, что максимальный размер подмножества с суммой, не превосходящей x строго меньше, чем минимальный размер подмножества с суммой больше x, то число x удачное, а иначе нет. Получим решение за $O(qc^3)$.

Введем обозначение $S = \{a_1, a_2, \ldots, a_n\}$, где $a_1 < a_2 < \ldots < a_n$.

Заметим, что решение первой подзадачи можно легко улучшить, если заметить, что для фиксированного k выполнено, что $s_{min} = a_1 + a_2 + \ldots + a_k$ и $s_{max} = a_{n-k+1} + \ldots + a_n$. Таким образом, имея множество S, мы можем получить все полуинтервалы $[s_{min}, s_{max})$, перебрав все k за время $O(|S| \log |S|)$. Далее мы можем легко получить длину их объединения за такое же время. В итоге получаем решение за $O((n+q)^2 \log (n+q))$.

Далее для того, чтобы решать задачу было удобнее будем считать количество удачных чисел от 0 до $sum(X) - 1$. Чтобы получить ответ на нашу задачу нам нужно просто из $sum(S)$ вычесть это число.

Пусть $0 \leqslant x < sum(X)$ — удачное. Заметим, что это равносильно тому, что $a_{n-k+1} + \ldots + a_n \leqslant x < a_1 + \ldots + a_{k+1}$ для некоторого $0 \leqslant k < n$. Это так, потому что если число удачное, то для некоторого размера k выполнено, что если подмножество будет размера $\leqslant k$, то сумма его элементов будет $\leqslant x$, а если подмножество будет размера $> k$, то сумма его элементов

будет $> x$. Максимальная сумма подмножества размера k это $a_{n-k+1} + \ldots + a_n$, минимальная сумма подмножества размера $k+1$ это $a_1 + \ldots + a_{k+1}$.

Значит, чтобы искать нужное нам количество удачных чисел, нам нужно найти длину объединения полуинтервалов $[a_{n-k+1} + \ldots + a_n, a_1 + \ldots + a_{k+1})$ по всем $0 \leqslant k < n$. Заметим, что некоторые из этих полуинтервалов пустые. Заметим, что непустые полуинтервалы не пересекаются. Это очевидно так, потому что правая граница любого полуинтервала не больше левой границы следующего полуинтервала $a_1 + \ldots + a_{k+1} \leqslant a_{n-k} + \ldots + a_n$. Значит ответ на задачу это просто сумма длин этих полуинтервалов. Таким образом мы получаем, что то число, которое мы ищем, это $\sum\limits_{k=0}^{n-1} \max(a_1 + \ldots + a_{k+1} - a_{n-k+1} - \ldots - a_n, 0)$.

Значит мы можем решить третью подзадачу по-другому за время $O((n+q)^2)$. Будем за линейное время поддерживать массив a и считать нужную нам сумму.

Обозначим $f(k) = a_1 + \ldots + a_{k+1} - a_{n-k+1} - \ldots - a_n$. Заметим несколько простых утверждений:

- $f(k) = f(n-1-k)$
- $f(x) \geqslant f(y)$, при $x \leqslant y \leqslant \frac{n-1}{2}$

Решим задачу, пользуясь этими утверждениями. Во первых будем искать $\sum\limits_{k=0}^{\frac{n-1}{2}} \max(f(k), 0)$. Получить нужную нам величину из этой очень просто — нужно умножить на 2 и возможно вычесть число посередине, если n — нечетно.

Заметим, что в сумме $\sum\limits_{k=0}^{\frac{n-1}{2}} \max(f(k), 0)$ по второму утверждению несколько первых слагаемых > 0, а следующие $= 0$, потому что в максимуме начнет превосходить 0.

Найдем бинарным поиском минимальный момент $l \leqslant \frac{n-1}{2}$, такой что $f(l) \leqslant 0$. Тогда наша задача после этого будет в том, чтобы вычислить $\sum\limits_{k=0}^{l-1} f(k) = \sum\limits_{k=0}^{l-1} (a_1 + \ldots + a_{k+1} - a_{n-k+1} - \ldots - a_n) = \sum\limits_{i=0}^{l} a_i(l-i) - \sum\limits_{i=n-l+1}^{n} a_i(i+l-n)$.

Какие величины нам нужно научиться считать, чтобы решить задачу полностью?

- Сумму чисел на префиксе $\sum\limits_{i=0}^{k} a_i$. С помощью двух таких запросов к сумме на префиксе можно будет считать $f(k)$.
- Сумму числа умноженного на индекс на префиксе $\sum\limits_{i=0}^{k} a_i i$. С помощью этого вместе в первой величиной можно будет вычислять финальную сумму за несколько таких запросов.

Для того, чтобы отвечать на два этих запроса эффективно воспользуемся деревом отрезков. Изначально выпишем все числа, которые будут когда-либо в множестве S. Мы можем так сделать, потому что все запросы нам даны сразу. На этом массиве будем поддерживать дерево отрезков, где в каждой позиции будет находить число 0 или 1. Число 0 будет обозначать, что этого числа нет в текущем множестве S, 1 будет обозначать, что это число есть.

Для каждого отрезка этого ДО будем поддерживать 3 величины:

- количество чисел из отрезка, которые есть в текущем множестве S
- сумма чисел из отрезка, которые есть в текущем множестве S
- сумма чисел умноженных на индекс для чисел из отрезка, которые есть в текущем множестве S

Эти величины легко пересчитывать при сложении левого и правого сыновей отрезков в вершине ДО.

Чтобы вычислять две нужные нам суммы, будем делать спуск по нашему дереву отрезков, суммируя на том префиксе, количество чисел из S на котором равно k.

В итоге мы получаем решение за $O((n+q)\log^2(n+q))$, потому что мы делаем бинарный поиск, внутри которого обращаемся к дереву отрезков.

Такое решение без проблем должно было набирать полный балл. В случае использования в некоторых местах этого решения не самых оптимальных структур данных (например декартова дерева) или дополнительного бинарного поиска, решение должно получать от 55 до 100 баллов в зависимости от эффективности.