Задача	Разработка	Автор идеи
Прыжки по шкафам	Дмитрий Акулов	Дмитрий Акулов
Две люстры	Игорь Маркелов	Жюри коллективно
Игра на дереве	Николай Будин	Глеб Евстропов
Сортировка матрицы	Степан Стёпкин	Михаил Тихомиров
Поеду домой	Александр Курилкин	Александр Курилкин
Вставить текст	Ибрагим Мамилов и Филипп Грибов	Александр Курилкин и Глеб Евстропов
Плитка для ванной	Константин Амеличев	Михаил Пядёркин
Фокус с подмножествами	Иван Сафонов	Иван Сафонов

Председатель жюри:	Елена Владимировна Андреева
Директор соревнований:	Глеб Евстропов
Руководитель разработки задач:	Дмитрий Саютин
Московская методическая комиссия:	Елена Владимировна Андреева
	Глеб Евстропов
	Дмитрий Саютин
	Михаил Пядёркин
	Филипп Грибов
	Егор Чунаев
	Владимир Романов
	Григорий Резников
Авторы и разработчики задач,	Макс Ахмедов
не вошедших в финальный вариант:	Азат Исмагилов
	Алексей Перевышин
	Максим Деб Натх

Жюри также благодарит всех, кто участвовал в прорешивании и помог протестировать качество задач.

Михаил Ипатов	Никита Голиков	Павел Кунявский	Даниил Орешников
Арсений Кириллов	Максим Сурков	Андрей Ефремов	Иван Сафонов
Игорь Маркелов	Семён Степанов	Тимофей Василевский	Дмитрий Пискалов
Артём Захаренко	Дмитрий Ибраев	Дмитрий Акулов	Влад Носивской
Наталья Стрекаловская	Михаил Саврасов	Владимир Кауркин	Степан Стёпкин
Станислав Донской	Ибрагим Мамилов	Алексей Перевышин	Евгений Антрушин
Михаил Малютин	Николай Будин		

День 1

Задача OtVinta. Прыжки по шкафам

Идея 1

Можем заменить непосредственно высоты шкафов на разность между соседними: $d_i = h_{i+1} - h_i$. Как тогда получить w по d?
$w_i = \max(h_i, h_{i+1}, h_{i+2}) - \min(h_i, h_{i+1}, h_{i+2}) = \max(|d_i|, |d_{i+1}|, |d_i + d_{i+1}|)$

$O(nC)$

Сделаем динамическое программирование $dp[i][diff]$ - можно ли выбрать i первых разностей d_i, где последняя разность равна $diff$

Эту динамику можно считать за $O(nC)$, обозначим за $prev$ предыдущую разницу ($d_i = diff, d_{i-1} = prev$. Рассмотрим как вычислить состояние $dp[i][diff]$ и какие $prev$ стоит рассмотреть.

- Если $|diff| = w_i$ и $sign(prev) \neq sign(dif)$, то должно быть $|prev| \leqslant w_{pref}$. Таких значений $prev$ много, но с помощью префиксных сумм можно обработать такие переходы за $O(1)$ для каждой $diff$.
- Если $|diff| \leqslant w_i$ и $sign(prev) \neq sign(diff)$, то требуется $|prev| = w_{pref}$
- Если $|diff| \leqslant w_i$ и $sign(diff) = sign(prev)$, то $|diff + prev| = w_i$

Note 2

Если мы можем получить разность $diff$, то и $-diff$ мы можем получить тоже, так как, можно инвертировать все d_i. Будем учитывать только $diff \in [0, C]$

$O(n^2)$

Давайте хранить для каждого слоя значения, которые мы можем получить, не явно, а отрезками. Покажем, что на каждом слое их будет $O(n)$.

- Если на прошлом слое было значение w_i, то на этом мы можем получить отрезок: $[0; w_i]$
- Каждый отрезок $[l, r]$ с прошлого слоя даст нам отрезок $[\max(w_i - r, 0), w_i - l]$
- Если хотя бы один отрезок содержал значение $x : x \leqslant w_i$, то добавится отрезок: $[w_i, w_i]$

Итого, количество отрезков с переходом к новому слою увеличивается не более чем на 1.
Восстановление ответа: Так как на каждом слое мы храним все отрезки, мы можем проверить, есть ли на прошлом слое:

- w_i
- $w_i - diff$
- кто-угодно $\leqslant w_i$, если $diff = w_i$

Разностям надо вернуть знаки, раз мы стали рассматривать только положительные значения. Можем идти слева направо и раздавать их жадно, выбирая подходящий

$O(n)$

Отрезки еще и не сильно меняются. Каждое изменение слоя:

- инверсия + сдвиг + усечение нулем ($[l, r] \to [\max(w_i - r, 0), w_i - l]$)
- проверка наличия крайнего значения ($w_i \to [0, w_i]$)
- добавление крайнего значения ($x \leqslant w_i \to [w_i, w_i]$)

Все, кроме инверсии и сдвига происходит с краю нашего списка отрезков(храним их отсортированными), а инверсия и сдвиг одинаковы для всех отрезков. Поддерживаем $cf = \pm 1$ и сдвиг sh, пересчитывая их на каждом слое, и не изменяя отрезки непосредственно
Пересчитав значения cf и sh: сначала обрежем отрезки с краев, чтобы хранить только значения внутри $[0, w_i]$, после чего добавим $[w_i, w_i]$(преобразованный), если нужно. Количество урезаний будет $O(n)$.
Восстановление:

- Если было w_i в прошлом слое - запоминаем и берем
- Если $diff = w_i$, берем минимум
- Иначе мы переходим гарантированно в $w_i - diff$

Задача PartyLikeARussian. Две люстры

Формальная постановка задачи звучит так: есть две последовательности различных чисел, бесконечно зацикленных. Требуется посчитать префикс минимальной длины, на котором есть ровно k позиций i, таких, что $a_i \neq b_i$.

Для решения первой подзадачи достаточно просто промоделировать процесс. Это займёт $O((n+m) \cdot k)$ времени.

Для решения второй подзадачи требовалось заметить, что если научиться считать считать на префиксе количество не совпадающих позиций, то можно применить бинарный поиск по ответу. Немного удобнее будет вместо этого считать на префиксе число совпадающих позиций, а число не совпадающих на этом же префиксе легко выражается через это.

Так как все числа в одной последовательности различные, нужно научиться считать для каждого числа, встречающегося в обеих последовательностях, количество неотрицательных решений системы сравнений:

$$\begin{cases} pos \equiv x \pmod{n} \\ pos \equiv y \pmod{m} \end{cases}$$

Где x и y позиции этого числа во входных последовательностях.

Это можно сделать с помощью Китайской Теоремы об Остатках и несложных формул. Асимптотика такого решения $O((n+m) \cdot \log(n+m) \cdot \log(k \cdot (n+m)))$.

Опишем кратко как это делать, и заодно поймём как это сделать эффективнее для нашей задачи.

Во-первых если $x \bmod \gcd(n,m) \neq y \bmod \gcd n, m$, то таких pos заведомо нет. В противном случае достаточно научиться решать систему уровнений

$$\begin{cases} pos' \equiv x' \pmod{n/gcd(n,m)} \\ pos' \equiv y' \pmod{m/gcd(n,m)} \end{cases}$$

Где x' равен целой части x от деления на $\gcd(n,m)$, а y' равен целой части от деления y на $\gcd(n,m)$. Итого мы свели задачу к случая взаимнопростых n, m.

Можно показать, что решением такой системы является

$$pos \bmod n \cdot m = (x \cdot a \cdot m + y \cdot b \cdot n) \bmod n \cdot m$$

Где $a \cdot m = 0 \bmod n$, а $b \cdot n = 0 \bmod m$. Такие a и b заведомо существуют из теоретико-численных соображений. В контексте нашей задаче их можно заранее предподсчитать за линейное время, и использовать для решения всех уравнений.

Получится решение за $O((n+m) \cdot \log((n+m) \cdot k))$.

Задача TheWinnerTakesItAll. Игра на дереве

Заметим, что первый игрок первым ходом обязательно должен поставить свою фишку ровно в центр диаметра. Иначе, второй игрок может поставить свою фишку в соседнюю вершину в направлении центра, и тогда первый игрок проиграет. Значит мы знаем первый ход первого игрока, и можем считать, что диаметр имеет чётную длину.

Переберем вершину, в которую второй игрок поставит свою фишку. Теперь процесс игры можно довольно просто просимулировать. Своим ходом первый игрок может пойти не в направлении второго, тогда можно независимо вычислить для каждого игрока максимальное расстояние, которое он сможет пройти. Первому выгодно так сделать, если он при этом выиграет, то есть если его максимальное расстояние строго больше, чем максимальное расстояние второго игрока. Иначе, первому игроку ничего не остается, кроме как пойти в направлении второго игрока. Для второго игрока аналогично: он может либо пойти не в направлении первого, он так сделает, если в результате выиграет, либо он пойдет в направлении первого.

Используя эту идею можно написать решение за $\mathcal{O}(n^2)$ или $\mathcal{O}(n^3)$ (в зависимости от эффективности подсчёта описанных выше расстояний) — перебрать первый ход второго и симулировать процесс игры как описано выше. Решим быстрее.

Обозначим расстояние от стартовой вершины первого игрока до стартовой вершины второго игрока за d. Пронумеруем вершины на пути от стартовой вершины первого до стартовой вершины второго от 0 до d. Обозначим за a_i максимальное расстояние, которое можно пройти, если свернуть с пути в вершине i.

Тогда первый свернет с пути на i-м ходу (первое перемещение фишки соответствует ходу 0), если:

$$a_i > \max_{j=i+1}^{d-i}((d-i)-j) + a_j$$

Второй свернет с пути на i-м ходу, если:

$$a_{d-i} > \max_{j=i+1}^{d-i-1}(j-(i+1)) + a_j$$

Обозначим $b_i = a_i + i$ и $c_i = a_i - i$. Тогда первое условие можно преобразовать как:

$$b_i > \max_{j=i+1}^{d-i} c_j + d$$

А второе как:

$$c_{d-i} + d \geqslant \max_{j=i+1}^{d-i-1} b_j$$

Будем обходить дерево из центра dfs-ом, будем считать что второй игрок изначально ставит свою фишку в ту вершину, где сейчас находится dfs. Тогда последовательность a довольно просто поддерживать. При переходе в ребенка в dfs-е, в последовательности a изменяется последний элемент и добавляется еще один элемент в конец.

Отсюда следует решение за $O(n^2 \cdot \log(n))$. Будем хранить b и c в структуре данных, которая позволяет изменять значение элемента и находить максимум на отрезке (например, дерево отрезков). Тогда для каждой стартовой вершины второго игрока, можно провести симуляцию игры за $O(n \cdot \log(n))$.

Вместо симуляции, можно найти минимальное i ($i \leqslant \frac{d}{2}$), в котором выполняется условие на выигрыш первого игрока. И минимальное j ($j < \frac{d}{2}$), что в $d-j$ выполняется условие на выигрыш второго игрока. После чего, выигрывает тот игрок, у которого найденное число меньше. Обратите внимание, что одно или оба числа могут не существовать, но эти случаи также разбираются.

Заметим, что $\sum_{i=0}^{d} a_i \leqslant n$. А также, заметим, что если $a_i < (\frac{d}{2} - i) \cdot 2$, то первый игрок точно не выиграет, если свернет в вершине i, потому что второму достаточно будет продолжить идти по пути, чтобы выиграть. Аналогично для второго игрока. Воспользуемся корневой декомпозицией. Будем

запоминать индексы позиций, в которых $a_i > Q$. Тогда для первого игрока нужно проверить их, а также позиции $\frac{d}{2} - \frac{Q}{2} \leqslant i \leqslant \frac{d}{2}$. Аналогично для второго игрока. Значит, проверка будет работать за $O((\frac{n}{Q} + Q) \cdot \log(n))$. Если выбрать $Q = \sqrt{n}$, то итоговое время работы будет $O(n \cdot \sqrt{n} \cdot \log(n))$, однако решение имеет очень хорошую константу.

Задача Bukhgalter. Сортировка матрицы

Если $A = B$, выведем 0. Далее считаем, что $A \neq B$

Заметим, что каждый столбец имеет смысл сортировать не более одного раза (можно оставить только последнюю сортировку с ним связанную).

1 подгруппа — $O(2^m \cdot m! \cdot nm)$ Давайте переберем всевозможные 2^m подмножеств столбцов, а потом для каждого подмножества переберем порядок в котором мы будем сортировать столбцы и посмотрим что получиться.

2 подгруппа Заметим, что если B можно получить, то в B найдется столбец вида 1, 2, ... n, а значит этот столбец можно отсортировать в A, и так как в нем все элементы различны, то A станет равна B.

3 подгруппа Оказывается, что ответ можно перебирать чуть умнее — будем перебирать перестановки, а также сколько элементов из перестановки будет использовано. $O(m! \cdot m \cdot nm)$

5,6 подгруппа

Идея — выберем столбец, который мы отсортируем последним. Конкретный столбец можно отсортировать последним, если нет двух строк, которые из-за этого станут упорядочены неправильно. Также заметим, что такое действие нам не помешает получить ответ. Будем продолжать выбирать предпоследний, предпредпоследние столбцы, etc. В результате строки будут разбиваться на группы (классы эквивалентности по элементам в фиксированных столбцах). В чем их смысл? Если две строки находятся в одном классе, то в столбцах которые мы отсортировали, в соответствующих строках были одинаковые значения. Тогда если мы будем находить столбец, сортировка которого не нарушает порядок между строками внутри каждого из классов, то его можно безопасно отсортировать. Давайте каждый раз перебирать такой столбец. Нам не важно если мы отсортируем какой-то лишний столбец — из-за условия возможности применения его в самом конце, точно ничего не испортит. Тогда просто сортируем столбец, если могли. Каждый раз количество классов эквивалентности не убывает, поэтому и количество столбцов которые можно отсортировать ничего не испортив не уменьшается. Если после всех применений отсортировать не получилось, значит что получить из матрицы A матрицу B невозможно.

В зависимости от реализации $O(nm^2log)$ или $O(nm^2)$ решение проходит 5 или 6 подгруппу.

Решение на полный балл - $O(nm)$

Давайте не хранить явно классы эквивалентности. Для этого заметим, что в таблице B, классы эквивалентности соответствуют подрезкам строк матрицы B! Тогда нам достаточно хранить для каждой пары соседних строк, смогли ли мы их разделить (свапнуть нижний и верхний местами) — назовем этот массив *cansplit*. Также, для каждого столбца будем хранить количество еще неразрешенных инверсий (здесь и далее мы смотрим на инверсии между соседними элментами), назовем его *cnt*. Изначально — это просто количество инверсий в столбце. Далее, это будет то же количество инверсий, но только внутри каждого из классов. Чем это решение отличается от куба? Мы умеем быстро находить какой столбец можно применить - это столбец j для которого $cnt[j] = 0$. Тогда нам осталось только научиться обновлять cnt. Давайте поддерживать очередь, в которой будут только столбцы j для которых $cnt[j] = 0$, применять сортировку по столбцу, обновлять cnt и добавлять новые столбцы для которых cnt занулилось. В этом нам поможет *cansplit*. Давайте, если у нас получилось поменять строки местами (то есть в рассматриваемом сейчас столбце v выполнено $b[i][v] < b[i-1][v]$), запишем $cansplit[i] = 1$ и обновим все значения *cnt* учитывая возможность обмена данной пары строк. Понятно, что дважды разделять строки не имеет смысла, поэтому если $cansplit[i] = 1$ это означает, что через данную пару мы уже обновили все *cnt* и добавлять ее в очередь во второй не надо (можно заметить, что этот процесс похож на поиск в ширину по зануляющимся элементам cnt). Так как каждый столбец мы применим не более одного раза, и пар соседних строк $O(n)$, то решение работает за $O(n \cdot m)$. Не забываем, что не обязательно нужно разбить на ровно n

классов эквивалентности, ведь и меньшего количества может хватить. Поэтому в конце проверим наши преобразования(не сложно, но нужно быть аккуратным).

Также существует решение за $O(nmlog)$, но, по мнению жюри, оно немного сложнее и в плане понимания, и в плане реализации.

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.