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.