Задача A. Разбиение на слагаемые

Если число можно разбить на k различных слагаемых, то их сумма хотя бы $1+2+\ldots+k = \frac{k(k+1)}{2}$. Действительно, минимальное уникальное число не меньше одного, второе — двух и так далее.

Научимся строить пример, когда $n \geqslant \frac{k(k+1)}{2}$. Выпишем числа от 1 до k и добавим к максимальному числу то, что осталось распределить, то есть $n - \frac{k(k+1)}{2}$. Все числа останутся различными и их сумма будет равна n, что и требовалось.

Задача B. Нетреугольники

Самое простое решение работало за $O(n^3 \cdot q)$ и набирало 10 баллов. Достаточно было поддерживать рёбра (например в матрице смежности), и после каждого запроса проверять все n^3 троек.

Дальше можно было заметить, максимальная стоимость нетреугольника — на самом деле максимальная стоимость трёх вершин, где первые две не соединены ребром. Поэтому третью вершину всегда выгодно взять наибольшей возможной, при этом она не должна совпадать с первыми двумя. Поэтому такую вершину можно найти за 3 действия среди трёх вершин с наибольшей стоимостью, так что теперь мы можем перебирать только пары вершин. Такое решение работало за $O(n^2 \cdot q)$ и набирало 20 баллов.

Можно было сохранить все подходящие пары вершин (и максимальные нетреугольники для них) в какой-нибудь структуре данных. Такие решения набирали от 20 до 55 баллов в зависимости от реализации.

В подгруппе на 20 баллов можно было заметить следующий факт: ответ не уменьшается (так как рёбра только удаляются). При этом для каждой вершины v, нас интересует только максимальная по стоимости вершина u, с которой v не соединена ребром, поэтому для каждой вершины мы можем перебрать все остальные в порядке невозрастания стоимостей, и найти первую подходящую. После того, как мы нашли ответ для исходной задачи, надо было лишь увеличивать его после каждого добавления. Для этого достаточно было считать ответ для удалённого ребра. Такое решение работало за $O(n \log n + m + q)$ (с учётом сортировки вершин по стоимости).

Чтобы получить 100 баллов, надо было совместить 2 предыдущих решения. Рассмотрим все ребра, и поделим их на 2 типа: рёбра, которые участвуют в запросах, и остальные. Рёбер первого типа не больше q. Для рёбер второго типа ответ не изменится, поэтому можно сразу посчитать для них ответ за $O(n + m + q)$. Рёбра первого типа можно поддерживать в какой-нибудь структуре данных, например, set. Всего удалений, добавлений и запросов минимума будет $O(q)$, поэтому итоговая асимптотика будет $O(n \log n + m + q \log q)$.

Задача C. Найти путь

Для начала заметим, что каждой вершине можно поставить в пару n вершин. Значит, первая вершина из пары будет иметь номер $v = \lceil \frac{k}{n} \rceil$. Теперь для нее нужно найти парную вершину, удовлетворяя условие на лексикографический порядок. Поскольку первая вершина зафиксирована, то теперь можно искать $((k-1) \mod n)$-й лексикографически путь из v (в 0-индексации).

Предположим, что вершина v не зависела от запроса (как в подгруппе с $k \leqslant n$). Тогда мы могли с самого начала подвесить дерево за вершину v, тем самым превратив дерево в корневое. После этого мы можем отсортировать исходящие ребра из каждой вершины по номеру вершины, в которое это ребро ведет.

Как теперь выглядит отсортированный список путей? Первый путь идет из корня в корень. Дальше сколько-то путей можно получить, если добавить в путь минимальное из ребер, после чего как-то продолжить путь в поддереве. После них — второе из минимальных ребер, и так далее. Список путей можно получить обходом в глубину (нам достаточно знать только последнюю вершину в пути):

```
dfs(v):
    paths.push(v)
    for (to : edges[v]):
        dfs(to)
```

Теперь можно насчитать для каждой вершины v массив $count_{to}$ — количество вершин, в которых можно закончить путь, если мы прошли по ребру $v \to to$. Теперь с помощью префиксных сумм мы

для каждого ребра насчитываем, каким должно быть k, чтобы из вершины v первый переход делался по этому ребру.

Вернемся к общей задаче. Пусть вершина v находилась в произвольном месте дерева. Заранее подвесим дерево за какую-то вершину и выпишем обход дерева относительно нее. Заметим, что для любой вершины v обход ее поддерева будет подотрезком общего обхода. Значит, если искомая вершина u лежала в поддереве вершины v, то ее можно найти за $O(1)$ обращением к нужному индексу обхода.

Но бывает так, что путь из v сначала идет на сколько-то ребер наверх, и только потом начинает идти вниз. Сделаем двоичные подъемы, чтобы узнать, насколько вверх нам надо переместиться. Будем считать, что нас всегда интересует задача $find(v, k)$ — найти k-й путь из вершины v. На текущий момент мы можем посчитать $find(v, k)$, если ответ лежит в поддереве вершины v.

Пусть $dp[v][x]$ — подотрезок значений k, при которых мы поднимаемся наверх на 2^x ребер. Как посчитать такую динамику? Для $x = 0$ она у нас уже посчитана ранее, осталось только понять, как объединить значения.

Заметим, что сначала k должно лежать в промежутке $dp[v][x-1]$, чтобы мы поднялись до вершины $parent[v][x-1]$. После этого из k вычитается то количество меньших последовательностей, которые мы отсекли, и нам надо, чтобы новое k попало в интервал $dp[parent[v][x-1]][x-1]$. Решив данные ограничения на k, мы получим новый промежуток для $dp[v][x]$.

Для ответа на запрос переберем степень двойки, на которую мы хотим подняться. После этого, если $k \in dp[v][x]$, то мы можем продолжить с $find(parent[v][x], k')$. В какой-то момент мы закончим подниматься наверх, нам останется сделать спуск вниз с помощью выписанного обхода.

Что надо не забыть:

- При склеивании $dp[v][x-1], dp[parent[v][x-1]][x-1]$ может так оказаться, что $dp[parent[v][x-1]][x-1]$ уже просуммировал размер поддерева, содержащего v. Чтобы это проверить, надо сравнить номера ребер вверх и вниз из вершины $parent[v][x-1]$. Если сын имел меньший номер, то к k надо прибавить размер поддерева.

- Аналогичная ситуация возникает при подъеме наверх при ответе на запрос.

- Когда мы нашли самую верхнюю вершину на пути и хотим взять элемент в ее поддереве, наша операция имеет вид $find(v, k)$. k может содержать в себе размер наддерева, если индекс предка был маленьким. Так что надо с помощью префиксных сумм проверить этот случай и, если k было велико, вычесть из него размер наддерева.

Задача D. Точки на плоскости

Первая, третья и пятая подзадачи решаются наивной реализацией ответов на запросы, нужно проверять, лежит ли точка в полуплоскости/квадрате со сторонами параллельными осям координат/произвольном квадрате. $O(n \cdot q)$

Для решения второй подзадачи построим выпуклую оболочку данного множества точек. Тогда нам нужно проверять, лежит ли хотя бы одна из вершин оболочки в полуплоскости. С помощью бинарного поиска найдём самую далёкую точку оболочки от прямой-направляющей полуплоскости, лежащую в нужной полуплоскости и проверим её. $O(n \log n + q \log n)$

Четвёртая подзадача — классическая задача на двумерную сканирующую прямую и дерево Фенвика. $O(n \log n + q \log n)$

В шестой и седьмой подзадачах нужно заметить, что любой квадрат на плоскости можно представить в виде 4 прямоугольных треугольников с катетами параллельными осям координат и 1 квадрата со сторонами, так же параллельными осям координат, для каждого квадрата выделим данные треугольники. Теперь мы свели исходную задачу к следующей: есть набор прямоугольных областей на плоскости (заданных границами по x: $[lx, rx]$ и по y: $[ly, ry]$) и полуплоскость для каждой такой области. Применим метод разделяй и властвуй($d\&c$) по координате x, заметим, что на каждом уровне рекурсии у нас O(n) точек и O(q) запросов. Для множества запросов, отрезок $[lx, rx]$ которых целиком покрывает текущий отрезок x в рекурсии применим divide and conquer по координате y. Заметим, что для $d\&c$ по y у нас так же, как и для $d\&c$ по x, линейное число точек и

запросов на каждом уровне рекурсии. Теперь, когда мы рассматриваем подпрямоугольник по x и по y, будем решать задачу для всех запросов, целиком покрывающих этот прямоугольник. Для этого воспользуемся решением второй подзадачи. $O((n+q)\log^3(n+q))$, если строить выпуклую оболочку за $O(n\log n)$ и отвечать на каждый запрос с помощью бинарного поиска, или $O((n+q)\log^2(n+q))$, если строить оболочку за $O(n)$, предварительно отсортировав точки и запросы и заменив бинарный поиск на метод двух указателей.

Решение для восьмой группы снова основывается на том, что мы умеем отвечать на на прямоугольной области, когда нас интересует только одна полуплоскость. Воспользуемся квадродеревом — будем делить области пополам (на квадраты), пока не можем ответить на запрос. В силу того, что запрос является квадратом, на каждом уровне деления будет O(1) областей с неотвеченными запросами. Если обрабатывать все запросы одновременно и воспользоваться методом двух указателей как в предыдущей группе, то можно получить асимптотику $O(n\log max\_coor)$.

Задача E. Игра на доске

Для решения задачи при $x \leqslant 10^5$ можно было промоделировать описанную игру.

Для полного решения необходимо заметить следующий факт: после первого раунда на доске обязательно появится число 0, так как количество очков проигравшего первый раунд игрока будет равно 0.

Заметим, когда 0 находится на доске, число Алисы всегда больше 0, а число Боба равно 0.

То есть все раунды, кроме самого первого, гарантированно выиграет Алиса.

Таким образом, осталось узнать, кто выиграет первый раунд.

Несложно понять, что Боб выиграет первый раунд только тогда, когда 0 изначально не было на доске.

Задача F. Наименьший общий предок

Для начала определим $f(m)$ как число подмножеств, наименьший общий предок которых равен m. Тогда ответ будет равен $1 \cdot f(1) + \ldots + n \cdot f(n)$. Далее будем вычислять $f(m)$.

Пусть произвольная вершина k из подмножества не лежит в поддереве m. Тогда m не станет наименьшим общим предком, поскольку m не лежит на простом пути от 1 до k. Тогда далее будем рассматривать только подмножества вершин в поддереве вершины m.

Вместо исходной задачи решим противоположную и посчитаем число «плохих» подмножеств, которые содержатся в поддереве m, наименьший общий предок которых не равен m, и с помощью этого найдём $f(m)$.

Для этого заметим, что если путь между двумя вершинами в поддереве проходит через m, то их наименьший общий предок равен m. Тогда это означает, что каждое «плохое» подмножество содержится в поддереве некоторого сына m_i. Также заметим, что если некое подмножество содержится в поддереве m_i, то оно «плохое», так как наименьший общий предок будет лежать в поддереве m_i.

Для удобства определим $size(x)$ как размер поддерева у вершины x, $m_1, m_2, \ldots, m_k$ — сыновья m. Нетрудно заметить, что число непустых подмножеств в поддереве x равно $g(x) = 2^{size(x)} - 1$. С использованием рассуждений выше мы получаем, что количество «плохих» подмножеств будет ровно $g(m_1) + g(m_2) + \ldots + g(m_k)$.

Значит остальные подмножества дают нужный наименьший общий предок. Так как мы знаем, что количество непустых подмножеств $g(m)$, то $f(m) = g(m) - g(m_1) - g(m_2) - \ldots - g(m_k)$.

Для подсчёта $size(x)$ требуется ДП на деревьях за $O(n)$. Для подсчёта $f(m)$ мы пройдёмся по каждому сыну ровно один раз, а их $O(n)$. Также заранее посчитаем все нужные степени двойки за $O(n)$. Тогда алгоритм работает за $O(n)$.

Задача G. Странная функция

Заметим, что p_i^{-1} — это номер позиции, на которой в перестановке p стоит элемент i. Поэтому, $|p_i^{-1} - p_{i+1}^{-1}|$ это расстояние между позициями элементов i и $i+1$ в перестановке p. А это равняется количеству элементов, расположенных между ними, увеличенным на единицу.

Обозначим за cnt_i количество элементов i в последовательности, а за all произведение всех cnt_i. Если перебрать позиции элементов i и $i+1$ в исходном последовательности, а также все вхождения

элемента j между ними, не равный им, то эта тройка элементов лежит в $\frac{all}{cnt_i \cdot cnt_{i+1} \cdot cnt_j}$ перестановках. В каждой из этих перестановках, элемент j увеличит расстояние между i и $i+1$ на 1, а значит ровно на столько надо увеличить ответ. В каждой перестановке, мы не учли исходное расстояние между элементами i и $i+1$, и поэтому ответ надо увеличить на $all \cdot (m-1)$. Получилась асимптотика $O(n^3)$.

Посмотрим на элемент j. Для него интересно узнать, сколько существует пар элементов i и $i+1$, что они не равны j и лежат по разные стороны от него. Заведём массивы $left_i$ и $right_i$ — количество элементов, равных i, слева и справа от текущего элемента соответственно. А также, будем поддерживать суммы $\frac{all \cdot (left_i \cdot rigth_{i+1} + left_{i+1} \cdot right_i)}{cnt_i \cdot cnt_{i+1}}$ по всем i. Тогда при сдвиге элемента j на один вправо, в массивах $left$ и $right$ изменяется $O(1)$ элементов, а в сумме изменяется $O(1)$ слагаемых. Для элемента j вычтем слагаемые, образованные парами $(j-1, j)$ и $(j, j+1)$ из этой суммы, и разделим оставшееся на cnt_j. Ровно для такого количества пар элемент j находится между ними, и поэтому ровно столько надо прибавить к ответу. Аналогично предыдущему решению, ответ надо увеличить на $all \cdot (m-1)$. Получилась асимптотика $O(n)$.

Задача H. Прибавление

Группа 1. $n, m, q \leqslant 100$

В первой группе можно явно поддерживать всё поле и выполнять все операции напрямую. В запросах первого типа честно обойдём каждый из t прямоугольников и прибавим в каждой клеточке по 1 – итого $O(nmt)$ на запрос. В запросах второго типа можем обойти весь заданный прямоугольник и найти сумму – итого $O(nm)$ на запрос. Итоговое время работы $O(nmqt)$, что нас устраивает, так как $t \leqslant m \leqslant 100$.

Группа 2. $n, m, q \leqslant 700$

Во второй группе мы все ещё можем поддерживать всё поле явно и обрабатывать запросы второго типа так же напрямую. Запросы первого типа надо обрабатывать хитрее, ведь мы не можем позволить себе t проходов по прямоугольникам. Для каждой клетки (r', c'), затронутой запросом, мы можем явно посчитать значение, которое в ней прибавится – это количество сдвигов исходного прямоугольника от 0 до $t-1$, покрывающих данную клетку. Оно равно $\min(c_{i_1} + t - 1, c') - \max(c_{i_1}, c' - (c_{i_2} - c_{i_1})) + 1$ (порисуйте такие отрезки на бумажке, чтобы понять, почему это так). Теперь мы можем обрабатывать запросы первого типа за один проход по затронутым клеткам, то есть за $O(nm)$.

Группа 3. $q \leqslant 10\,000$, $t_i = 1$

В третьей группе мы уже не можем явно поддерживать всё поле. Вместо этого будем искать ответ на запрос второго типа, считая отдельно для каждого предыдущего запроса первого типа, сколько мы прибавили на их пересечении. В этой группе $t_i = 1$, поэтому нам нужно посчитать суммарную площадь пересечения запроса суммы с каждым запросом изменения. Это можно сделать простыми формулами за $O(1)$, тогда суммарное время работы $O(q^2)$.

Группа 4. $q \leqslant 10\,000$

Четвёртая группа решается аналогично третьей, однако вместо простой площади пересечения здесь нужна другая формула, которая пригодится в полном решении. Рассмотрим запрос изменения. Так как сдвиги происходят по горизонтали, во всех строках $[r_{i_1}; r_{i_2}]$ в одинаковых столбцах будут прибавлены одинаковые значения, поэтому мы можем найти сумму в одной строке пересечения и умножить её на высоту пересечения.

Пусть в запросе первого типа приходит прямоугольник со столбцами l, r и параметром t. Тогда прибавление к t отрезкам $[l; r]$, $[l+1; r+1]$,…,$[l+t-1; r+t-1]$ можно представить как сначала поэлементное прибавление к суффиксу $[l; m]$ строки $(1, 2, \ldots, t, t, \ldots, t)$, а затем поэлементное вычитание из суффикса $[r+1; m]$ строки $(1, 2, \ldots, t, t, \ldots, t)$. Строка такого вида – это разность двух арифметических прогрессий $(1, 2, \ldots, t, (t+1)-1, (t+2)-2, \ldots)$, первая из которых прибавляется начиная с первого элемента, а вторая вычитается начиная с $(t+1)$-го элемента.

Получается следующее: мы прибавляем две такие прогрессии, одну начиная с l-го элемента и вторую начиная с $(r+t+1)$-го элемента, а затем вычитаем две такие прогрессии, одну начиная

с $(l+t)$-го элемента и вторую начиная с $(r+1)$-го элемента. Простой формулой можно посчитать сумму любого подотрезка арифметической прогрессии, поэтому по имеющемуся отрезку пересечения запроса суммы и изменения мы можем выяснить, кусочки каких из 4 прогрессий попадают в это пересечение, и посчитать их суммы за $O(1)$. Итого на каждый запрос отвечаем за $O(q)$, суммарное время работы $O(q^2)$.

Технический момент: модуль 2^{31} в условии дан не случайно – при больших n, m, q постоянно возникают переполнения, при этом в формулах надо делить на число 2, для которого не существует обратного ни по какому модулю 2^k. Если производить вычисления по модулю 2^{32} (переполнение 32-битного беззнакового целого типа) и делить на 2 после вычисления числителя, то мы получим как раз искомый остаток по модулю 2^{31}.

Группа 5. В запросах типа 2 сумма берётся по всему полю.

Самая простая группа — поддерживаем переменную, в которой будет содержаться сумма по всем клеткам поля, тогда после запроса первого типа к ней прибавляется St, где S – площадь прямоугольника запроса.

Группа 6. $n, m \leqslant 1000$, $t_i = 1$, запросы суммы после запросов изменения.

Группы 6 и 7 содержат ключевую идею полного решения. Назовём префиксом (r, c) поля все клетки (r', c') такие, что $r' \leqslant r$ и $c' \leqslant c$. Аналогично, назовём суффиксом (r, c) поля все клетки (r', c') такие, что $r' \geqslant r$ и $c' \geqslant c$. Если найдём сумму клеток на префиксе каждой клетки, то сможем за $O(1)$ найти сумму на любом прямоугольнике – она выражается через суммы на четырёх префиксах.

Запрос изменения можно представить в виде четырёх запросов прибавления на суффиксах поля. При этом если мы сделали прибавление на суффиксе (r_0, c_0), то сумма на префиксе (r, c) (где (r, c) лежит на суффиксе (r_0, c_0)) равна $(r - r_0 + 1)(c - c_0 + 1)$. Таким образом, мы прибавляем в клетке (r_0, c_0) функцию от двух переменных r и c, $(r - r_0 + 1)(c - c_0 + 1)$, и сумма на конкретном префиксе (r_1, c_1) равна значению этой функции в точке (r_1, c_1), если (r_0, c_0) лежит на этом префиксе (и равна 0, если не лежит).

Тогда в каждой клетке поля мы можем поддерживать функцию, прибавленную в ней, и после всех запросов изменения построить таблицу F, где $F[r][c]$ равно сумме всех функций в клетках префикса (r, c). После этого сумма на префиксе исходного поля (r, c) равна значению функции $F[r][c]$ в точке (r, c). Итоговое время работы — $O(nm + q)$.

Группа 7. $n, m \leqslant 1000$, запросы суммы после запросов изменения.

Идея такая же, как в группе 6, мы опять будем прибавлять и суммировать функции на префиксах. В седьмой группе функция более сложная, здесь нужно применить формулы из группы 4, представив запрос изменения как прибавление и вычитание арифметических прогрессий начиная с разных столбцов. Прибавлять и вычитать можно функции вида $\frac{(c-c_0+1)(c-c_0+2)}{2}(r - r_0)$, тогда запрос изменения сводится к прибавлению и вычитанию прогрессий в четырёх строчках в строке r_{i_1} и прибавлению со сменой знака в тех же столбцах в строке $r_{i_2} + 1$.

Полное решение (группы 8 и 9)

Полное решение требует более хитро поддерживать таблицу $F[n][m]$, так как размеры поля не позволяют это делать явно. Мы хотим уметь прибавлять функцию в точке и считать сумму функций на префиксе. Можно использовать структуру данных двумерное дерево отрезков, построенное неявно, но оно слишком сложное в написании и медленное. Вместо этого воспользуемся тем фактом, что запросы можно обрабатывать offline.

Авторское решение использует более быструю структуру данных — двумерное дерево Фенвика. Конечно, мы не можем построить его напрямую, даже сжав координаты, поэтому мы оставим только интересные точки. В качестве интересных точек выберем координаты вершин прямоугольников, которые нам пришли в запросах суммы. Внешнее дерево построим на всех интересных строках, и для каждого запроса суммы добавим соответствующие столбцы во все интересные строки, которые мы посетим во время спуска по внешнему дереву Фенвика при ответе на данный запрос. Тогда в запросах изменения (прибавления в точке) во внешнем дереве мы выберем в качестве начальной

самую нижнюю строку, лежащую не ниже данной точки, а в каждом внутреннем дереве в качестве начального будем выбирать самый левый столбец, лежащий не левее данной точки.

Итого, сложность по памяти $O(q \log q)$, а суммарное время работы $O(q \log^2 q)$ с очень большой константой, так как на хранение каждой функции требуется 6 чисел, и на каждый запрос изменения нужно 8 прибавлений в дереве Фенвика, однако благодаря его скорости решение работает довольно быстро.