

Задача А. Странная сумма

Для первой подзадачи достаточно было перебрать первые k пар $a_i \oplus b_i$, так как k достаточно мало, чтобы алгоритм со сложностью $O(k)$ работал в пределах заданного временного ограничения.

Вторая и третья подзадачи решаются путём сведения задачи к циклической последовательности длины m , где $c_i = a_1 \oplus b_i$ (или $c_i = a_i \oplus b_i$) для каждого $i = 1, 2, \dots, m$. Мы замечаем, что искомая сумма равна $\sum_{i=1}^k c_i$, которую можно относительно легко вычислить.

Чтобы получить полный балл за задачу, полезно заметить, что эта сумма независима по битовым разрядам, т.е.:

$$\sum_{i=1}^k (a_i \oplus b_i) = \sum_{j=0}^{\infty} \left(\sum_{i=1}^k (a_{i,j} \oplus b_{i,j}) \right) \cdot 2^j,$$

где $a_{i,j}$ обозначает битовый разряд, соответствующий степени 2^j , и аналогично для $b_{i,j}$. Это упрощает задачу, сводя её к массивам, состоящим из 0 и 1.

Теперь для каждого элемента первого массива мы будем суммировать все пары побитового исключающего ИЛИ, в которых этот элемент встречается. Благодаря сведению к новым массивам, это фактически сводится к суммированию всех элементов, с которыми он будет спарен, или суммированию их дополнений. Единственная оставшаяся задача — определить эти элементы и вычислить их сумму.

Для этого необходимо ключевое наблюдение. Пусть $g = \text{НОД}(n, m)$ — наибольший общий делитель n и m . Для очень больших чисел k требуемые элементы второго массива, для фиксированного элемента первого массива, образуют цикл длины $\frac{m}{g}$. Кроме того, общее количество различных циклов (с точностью до циклического сдвига) будет равно g . При эффективной реализации этого алгоритма, достигающей сложности $O((n + m) \log m)$, можно получить полный балл за эту задачу.

Задача В. Странная сортировка

Переформулируем задачу: от нас требуют считать инверсии на массиве, в котором сравнение происходит по правилу $a_i < a_j \Leftrightarrow a_i$ стоит раньше чем a_j в целевой перестановке. Вспомним как определяются инверсии на массиве:

$$\sum_{i=1}^n \sum_{j=i+1}^n [a_i > a_j]$$

Где $[x]$ — равно 1 если выражение истинно, и 0 если ложно. Что значит операция обмена двух соседних элементов p_j и p_{j+1} в целевой перестановке? Это значит что все выражения $[p_i > p_j]$ меняют свое значение, то есть если они были 0, то становится 1 и наоборот. Посчитаем изначальные инверсии на массиве inv .

Будем отвечать на запросы offline, то есть сначала считаем их, потом в каком-либо порядке ответим и выведем ответы на запросы. Тогда мы можем просимулировать перестановку на i запрос и понять, у какой пары значений их условия поменяются. Пусть это была пара x, y , тогда нам достаточно посчитать количество условий которые равны 1 или количество условий которые равны 0, потому что мы знаем что все утверждения либо 1 либо 0, а всего утверждений $cnt_x \cdot cnt_y$ — где cnt_i количество вхождения числа i в изначальный массив a . Откуда зная какое-то из значений $cond_0, cond_1$ мы можем вычислить второе, потому что $cond_0 + cond_1 = cnt_x \cdot cnt_y$. Посмотрим как изменится сумма: $cond_0 \cdot 0 + cond_1 \cdot 1 \rightarrow cond_0 \cdot 1 + cond_1 \cdot 0$. Зачем все это нужно? Давайте насчитаем какое-то из $cond_0, cond_1$ в зависимости от того что меньше cnt_x или cnt_y . То есть хотим для каждой пары значений (x, y) насчитать $cond_0$ или $cond_1$ за $O(\min(cnt_x, cnt_y))$, потом мы покажем почему нам этого достаточно.

Чтобы насчитать для пары $(x, y), cnt_x \leq cnt_y$ ответ будем обрабатывать массив слева направо и хранить текущие количества вхождения каждого значения $pref_i$. Запомним заранее для x для каких y мы хотим посчитать значения, после чего когда будем встречать в массиве x пройдемся честно по всем запросам y и насчитаем ответы. Тогда мы для каждого x отработаем для каждого запроса за cnt_x , осталось оценить данное решение. Отсортируем массив cnt : понятно, что самый худший случай это когда каждый следующий запрос будет работать за максимально возможное

cnt , и казалось бы тогда наш алгоритм будет работать за $O(nq)$, но давайте посмотрим сколько пар y может существовать для конкретного x , не более чем $\lfloor \frac{n}{cnt_x} \rfloor$, иначе сумма всех cnt была бы больше n , что невозможно. Сколько мы сможем охватить различных x среди пар? Если мы охватили последние k иксов, то мы потратили тем самым $\frac{k(k-1)}{2}$ запросов, но для каждого x суммарно мы сделали не более $cnt_x \cdot \frac{n}{cnt_x} = O(n)$ действий, откуда так как $k \leq O(\sqrt{q})$ получаем асимптотику $O(n\sqrt{q})$, что заходит на полный балл.

Задача С. Странные руны

Рассмотрим структуру оптимальных решений для конкретного запроса q , в котором необходимо завершить путь в резервуаре e_q через s_q секунд. Охарактеризуем путь как упорядоченный список пар $(p_1, t_1), (p_2, t_2), \dots, (p_k, t_k)$, где (p_i, t_i) означает, что Бесси последний раз посетила резервуар p_i в момент времени t_i и собрала всю доступную ману. По определению, допустимый путь для запроса q должен заканчиваться в e_q (то есть $p_k = e_q$) и завершаться через s_q секунд (то есть $t_k = s_q$). Собранная мана в таком пути равна $\sum_{i=1}^k m_{p_i} \cdot t_i$. Конечно, некоторые пути могут быть недопустимыми. Формально, путь считается допустимым, если и только если $t_1 \geq 0$ и $t_{i+1} \leq t_i + D[p_i][p_{i+1}]$ для всех $i \in \{1, \dots, k-1\}$, где $D[a][b]$ представляет кратчайший путь от резервуара a до резервуара b (заметим, что D может быть предварительно вычислен с использованием алгоритма Флойда-Уоршелла за время $O(N^3)$).

С учётом этого, каков оптимальный допустимый путь для фиксированного набора $p_1, \dots, p_k$? Можно показать с помощью индукции или аргумента обмена, что оптимально ждать как можно дольше. Конкретно, это означает, что оптимальные времена t^* определяются следующим образом:

$$\begin{aligned} t_k^* &= s_q \\ t_{k-1}^* &= s_q - D[p_{k-1}][p_k] \\ t_{k-2}^* &= s_q - D[p_{k-1}][p_k] - D[p_{k-2}][p_{k-1}] \\ &\dots \\ t_1^* &= s_q - \sum_{i=1}^{k-1} D[p_i][p_{i+1}] \end{aligned}$$

Заметим, что если $t_1^* < 0$, то путь с резервуарами $p_1, \dots, p_k$ недопустим. Для первой подзадачи достаточно найти оптимальное решение, перебирая все $O(N!)$ возможные комбинации $p_1, \dots, p_k$ и вычисляя соответствующие $t_1^*, \dots, t_k^*$ для каждого запроса за общее время $O(N! \cdot (N + Q))$.

Вторая подзадача

Для второй подзадачи мы больше не можем позволить себе перебирать каждый запрос для каждого возможного пути. Мы ищем структуру, которая устраняет необходимость такого перебора. Для простоты изложения будем рассматривать только пути и запросы, заканчивающиеся в фиксированном резервуаре e , так как другие пути и запросы можно обрабатывать отдельно. На основе нашей формулы для t^* , путь будет допустимым и корректным для некоторого запроса, если и только если $\sum_{i=1}^{k-1} D[p_i][p_{i+1}] \leq s_q$. Более того, общее количество собранной маны будет равно:

$$\sum_{i=1}^k m_{p_i} \cdot t_i^* = \sum_{i=1}^k m_{p_i} \cdot \left(s_q - \sum_{j=i}^{k-1} D[p_j][p_{j+1}] \right) = \left(\sum_{i=1}^k m_{p_i} \right) \cdot s_q - \left(\sum_{i=1}^k \left(m_{p_i} \cdot \sum_{j=i}^{k-1} D[p_j][p_{j+1}] \right) \right).$$

Мы видим, что это выражение является линейной функцией от s_q . В частности, мы можем определить, что путь J допустим, если и только если $s_q \geq T_J$, и тогда путь J приносит ману $f_J(s_q) \triangleq \text{slope}_J \cdot s_q + \text{add}_J$, где:

$$T_J = \sum_{i=1}^{k-1} D[p_i][p_{i+1}]$$

$$\text{slope}_J = \sum_{i=1}^k m_{p_i}$$

$$\text{add}_J = \sum_{i=1}^k \left(m_{p_i} \cdot \sum_{j=i}^{k-1} D[p_j][p_{j+1}] \right)$$

Это сразу же подразумевает, что ответ на запрос q — это максимум над набором линейных функций:

$$\max_{J \text{ s.t. } T_J \leq s_q} f_J(s_q).$$

Если бы не условие $T_J \leq s_q$, запросы на максимум линейных функций в определённой точке можно было бы эффективно обрабатывать с использованием структуры данных, такой как Convex Hull Trick (CHT) или LineContainer. Чтобы учесть условие $T_J \leq s_q$, мы можем отсортировать запросы по s_q и пути по T_J , и постепенно добавлять линии в нашу структуру данных так, чтобы при ответе на запрос q пути с $T_J \leq s_q$ были точно в нашей структуре данных. Соответственно, мы можем перебрать все $O(N!)$ возможные комбинации $J = p_1, \dots, p_k$, вычислить соответствующие T_J и f_J , а затем ответить на запросы с использованием нашей структуры данных за общее время $O(N! \cdot N \cdot \log(N) + Q \log(Q))$ для второй подзадачи.

Оставшиеся подзадачи

Для оставшихся подзадач перебор всех $O(N!)$ возможных комбинаций $J = p_1, \dots, p_k$ слишком медленный. Мы делаем два структурных наблюдения:

1. Условие $T_J \leq s_q$ неожиданно неважно, так как:

$$\max_{J \text{ s.t. } T_J \leq s_q} f_J(s_q) = \max_J f_J(s_q).$$

Мы докажем это от противного. Предположим, что некоторый путь J' с $T_{J'} > s_q$ даёт оптимальное количество маны. Напомним, что:

$$f_{J'}(s_q) = \sum_{i=1}^k m_{p_i} \cdot \left(s_q - \sum_{j=i}^{k-1} D[p_j][p_{j+1}] \right).$$

Если $T_{J'} > s_q$, то слагаемое, соответствующее p_1 , должно быть отрицательным, и, следовательно, путь, соответствующий $p_2, \dots, p_k$, будет строго лучше. Таким образом, мы можем игнорировать ограничение T_J и просто оптимизировать $\max_J f_J(s_q)$.

2. Все пути с одинаковым неупорядоченным набором резервуаров $p_1, \dots, p_k$ имеют одинаковый наклон (см. определение slope_J выше). Более конкретно, пусть $\mathcal{P} \subseteq \{1, \dots, N\}$ обозначает некоторое подмножество резервуаров для посещения, и пусть P_J обозначает неупорядоченное множество резервуаров, которые посещает путь J . Все пути J с $P_J = \mathcal{P}$ удовлетворяют:

$$\text{slope}_J = \sum_{p \in \mathcal{P}} m_p.$$

Наше первое наблюдение показывает, что важны только slope_J и add_J (а не T_J). Наше второе наблюдение отмечает, что slope_J полностью определяется неупорядоченным набором резервуаров, которые посещает J . Комбинируя эти два наблюдения, мы заключаем, что можно оптимизировать по подмножествам резервуаров для посещения, и для каждого фиксированного подмножества $\mathcal{P}$ единственной значимой степенью свободы является add_J , поэтому мы можем просто рассмотреть линию с наилучшим add_J среди тех, которые посещают $\mathcal{P}$. Более формально:

$$\max_{J \text{ s.t. } T_J \leq s_q} f_J(s_q) = \max_J f_J(s_q) = \max_J \text{slope}_J \cdot s_q + \text{add}_J = \max_{\mathcal{P}} \left(\sum_{p \in \mathcal{P}} m_p \right) \cdot s_q + \left(\max_{J \text{ s.t. } P_J = \mathcal{P}} \text{add}_J \right).$$

Это выражение также является линейной функцией от s_q . Мы можем определить нашу оптимизацию как максимум линейных функций для каждого подмножества:

$$g_{\mathcal{P}}(s_q) \triangleq \text{slope}_{\mathcal{P}} \cdot s_q + \text{add}_{\mathcal{P}},$$

где:

$$\begin{aligned}\text{slope}_{\mathcal{P}} &= \sum_{p \in \mathcal{P}} m_p \\ \text{add}_{\mathcal{P}} &= \max_{J \text{ s.t. } P_J = \mathcal{P}} \text{add}_J\end{aligned}$$

Наша цель — оптимизировать $\max_{\mathcal{P}} g_{\mathcal{P}}(s_q)$. Остаётся только вычислить каждое $\text{add}_{\mathcal{P}}$. Мы вычисляем все $\text{add}_{\mathcal{P}}$ с помощью динамического программирования, где состояние $\text{dp}[\text{now}][\text{rem}]$ представляет множество rem оставшихся узлов для посещения (двигаясь назад по порядку) и текущее местоположение now . Напомним, что:

$$\text{add}_J = \sum_{i=1}^k \left(m_{p_i} \cdot \sum_{j=i}^{k-1} D[p_j][p_{j+1}] \right).$$

Соответственно, если мы решаем перейти к следующему местоположению $\text{next} \in \text{rem}$, то слагаемые в add_J для каждого $p \in \text{rem}$ увеличатся на $m_p \cdot D[\text{next}][\text{now}]$. Эта формулировка достаточна для разработки динамической программы с $O(2^N \cdot N)$ состояниями, $O(N)$ переходами и общей сложностью $O(2^N \cdot N^2)$.

Итоговое решение

Комбинируя все шаги, для каждого запроса мы выводим $\max_{\mathcal{P}} g_{\mathcal{P}}(s_q)$, где коэффициенты для $O(2^N \cdot N)$ линий $g_{\mathcal{P}}$ вычисляются с помощью динамического программирования за время $O(2^N \cdot N^2)$. Затем мы используем СНТ для вычисления $\max_{\mathcal{P}} g_{\mathcal{P}}(s_q)$ для каждого запроса. Существует некоторая гибкость в реализации. Например, можно отсортировать линии и запросы, а затем использовать вектор/дек для общей сложности $O(2^N \cdot N^2 + Q \log(Q))$ (как это сделано в коде ниже). Или можно подготовить структуру данных СНТ для линий заранее, а затем вычислить запросы без их сортировки, что даст общую сложность $O(2^N \cdot N^2 + QN)$.

Третья подзадача может быть решена с использованием вышеописанной динамической программы, но с перебором всех запросов (вместо использования СНТ), что даёт алгоритм с временной сложностью $O(2^N \cdot (N^2 + Q))$. Четвёртая и пятая подзадачи с $N = 16$ и $N = 17$ существуют для того, чтобы дать частичные баллы за слегка неоптимальные решения или решения, требующие оптимизации по константам.

Задача D. Странное НВП

Чтобы решить задачу за $O(n^3)$ или $O(n^2 \log n)$ заметим, что если мы зафиксировали i то если мы оставим на суффиксе только такие элементы j что $p_i < p_j$, то любая НВП из этого суффикса будет также и НВП начиная с позиции i , это верно и в другую сторону. Значит нам достаточно оставить данный суффикс и решить задачу для него независимо от i , тогда наша задача уметь определять для каждой позиции правда ли, что она содержится во всех НВП.

Насчитаем $\text{lis_prefix}_i, \text{lis_suffix}_i, \text{cnt_prefix}_i, \text{cnt_suffix}_i$ — длина НВП заканчивающаяся на позиции i , начинающаяся в позиции i и количество таких НВП. Эти данные можно предсчитать либо за $O(n^2)$ стандартной динамикой (для подсчета значений на suffix 'е достаточно делать ту же динамику в другую сторону) либо за $O(n \log n)$ с помощью дерева отрезков. Тогда элемент i встречается во всех НВП если $\text{lis_prefix}_i + \text{lis_suffix}_i - 1 = LIS$, $\text{cnt_prefix}_i \cdot \text{cnt_suffix}_i = CNT_LIS$, где LIS — размер НВП на всем массиве, CNT_LIS — количество НВП на всем массиве. Соответственно для суффикса мы отвечаем за $O(n^2)$ или $O(n \log n)$, суммарно получили $O(n^3)$ или $O(n^2 \log n)$.

Сделаем следующее наблюдение: если рассмотреть НВП начинающиеся в позиции i , то они будут выглядеть как какие-то пути на массиве, и то что число входит во все НВП значит что все пути проходят через этот элемент. Также если мы посмотрим на такой элемент j , что он является критическим для i , тогда и все элементы k критичные для j тоже критичны для i . Потому как все НВП проходят через j , но из предыдущего абзаца мы знаем что критические элементы не зависят от j , значит критические для j также содержатся и во всех НВП начиная с i . Но тогда нам достаточно найти самый левый критический элемент j , после чего будет выполняться $ans_i = ans_j + 1$.

Как найти такое j ? Будем идти по массиву p справа налево и насчитывать dp_i — максимальное НВП, начиная с позиции i . Мы хотим пересчитаться через такие $j > i$, что $p_j > p_i$ и dp_j максимально возможно. Пусть такое максимальное значение это k , найдем x, y — минимальная и максимальная позиция для которых $dp_x = dp_y = k$. Тогда пусть $x = y$, это значит что действительно следующий элемент в НВП мы можем выбрать единственным образом, а значит $j = x$. Пусть это не так, тогда рассмотрим какие-то НВП начинающиеся в x и y они будут оптимальными по условию, рассмотрим уже x', y' и x'', y'' — те же пары по тому же определению, но начиная уже с позиций x и y . Теперь если $x' = y' = x'' = y''$, то опять же все НВП будут проходить через x' , а значит $j = x'$, если это снова неправда будем продолжать так до того момента как не закончится НВП или не найдется нужное j . Заметим, что в нашем пути из x и y неважны промежуточные элементы, нам нужно найти просто найти первый момент, когда $x' = y'$, но если мы будем на это смотреть с элемента x , то это также значит что мы хотим для него найти такое же j , что и пытаемся найти сейчас для i . Тогда давайте будем параллельно строить следующий граф: пусть мы нашли для i нужное ему j тогда проведем ребро $i \rightarrow j$, таким образом мы будем строить корневое дерево, и для i предком будет j .

Тогда посмотрим на вершины x и y , они ведут в ближайшие справа критические, значит нам достаточно найти $lca(x, y)$, чтобы найти ближайшую справа критическую и для x и для y , а значит ближайшей критической для i . После чего $j = lca(x, y)$ и можно будет провести нужное ребро. Осталось реализовать данное решение: для этого будем достраивать бинарные подьемы, но мы теперь не можем пользоваться `tin`, `tout` для поиска LCA, но мы все еще можем считать LA, параллельно считая высоты. Тогда сделаем следующий алгоритм для вычисления LCA: пусть h_x, h_y — высоты вершин x, y соответственно, с помощью LA поднимем самую глубокую из этих вершин так, чтобы выполнялось $h_x = h_y$. Теперь будем прыгать параллельными бинарными подьемами. Так как мы находимся на одной высоте, то прыгнув на одинаковое значение вверх у нас могут быть две ситуации: мы прыгнули в одинаковые вершины, а значит мы находимся либо в LCA либо выше, иначе мы еще не допрыгнули до LCA. Тогда будем прыгать только тогда, когда прыжки не совпадают, после этого останется лишь подняться в предка если изначально не выполнялось $x = y$. Эта часть работает за $O(\log n)$.

Чтобы вычислить x, y будем параллельно с проходом хранить Дерево отрезков или Дерево Фенвика, в котором будем среди максимальных по dp искать минимальные позиции на отрезке $[p_i + 1, n]$. Эта часть также работает за $O(\log n)$. Итоговая асимптотика $O(n \log n)$, что заходит на полный балл.