Задача A. Степенные числа

Автор задачи и разработчик: Иван Пальченков

Идея этой задачи заключалась в том, чтобы заметить, что k-степенное число — это число, в записи которого в системе счисления с основанием k не содержится никаких цифр, кроме 0 и 1.

При $k_i = 2$ любое число при переводе состоит только из цифр 0 и 1, т.е. для решения **первой подзадачи** достаточно было вывести само число n. Что интересно, некоторые полные решения, реализованные недостаточно эффективно (с большой константой над асимптотикой времени работы), не проходили эту подзадачу по времени, поэтому, если вы сталкивались с такой проблемой, можно было отдельно поставить `if` на запросы с $k_i = 2$.

Для решения **второй подзадачи** можно заметить, что при $n \leqslant k$ ответом является n при $n = 1$, или k в противном случае. Действительно, число k всегда является k-степенным, потому что записывается как 10_k, и в принципе просто равно k^1, что подходит под определение.

В **третьей подзадаче** почти достаточно было написать последовательный перевод всех чисел больше либо равных n в систему счисления с основанием k, после чего для каждого из них проверить, является ли оно k-степенным. Для перевода числа в другую систему счисления можно воспользоваться стандартным алгоритмом:

1. вычислим последнюю цифру числа x как $x \bmod k$;
2. оставшиеся цифры образуют число $\left\lfloor \frac{x}{k} \right\rfloor$ — повторим шаг 1 для него, если оно больше 0;
3. в конце остается развернуть массив выписанных цифр.

Если заметить, что ответ всегда не превышает $n \cdot k$, потому что между n и $n \cdot k$ всегда есть число, равное k^a для некоторого целого a, то будет видно, что при $k \leqslant 50$ такое решение укладывается в ограничения по времени, а при $k > 50$ ответ будет состоять не более чем из четырех цифр в k-ичной системе счисления, и поэтому не более шестнадцати вариантов ответа можно просто перебрать отдельно.

Чтобы решить **четвертую и пятую подзадачи**, можно было предпосчитать все хорошие числа в интервале от 1 до 10^6 за $\mathcal{O}(10^6 \cdot \log_k n)$ описанным выше образом. Для нахождения первого большего либо равного хорошего числа для каждого запроса можно воспользоваться бинарным поиском по полученному массиву — в C++ это можно сделать, например, встроенной функцией `lower_bound`. Четвертая подзадача также не требовала перевода числа между системами счисления: достаточно было считать число как строку.

Шестая подзадача решалась перебором всех битовых масок длины $\log_k n$, то есть всех подмножеств тех степеней k, которые войдут в число как слагаемые. Действительно, если каждая цифра числа в k-ичной системе счисления равна 0 или 1, то у нас есть выбор для каждой степени k от 0 до $\log_k n$: взять ее в число или не взять.

Поскольку в этой подзадаче $k_i \geqslant 20$, то $\log_k n \leqslant 13$, то есть различных масок не более 2^{13}, что проходит в ограничения по времени с $q \leqslant 500$. Стоило обратить внимание на то, как из битовой маски получить десятичное число: проще всего предподсчитать степени k и просто сложить те, которые вошли в итоговый ответ.

Рекурсивный перебор в этой подзадаче работает эффективнее перебора двоичных масок и перевода в десятичную систему счисления, потому что можно считать получаемое в итоге перебора число по ходу перебора, не тратя на это лишние $\log_k n$ времени.

Наконец, **полное решение** — это стандартный алгоритм генерации следующего лексикографически комбинаторного объекта. Переведем число n в систему счисления с основанием k. Если в записи не встретилось цифр больше 1, само число n будет ответом. Иначе,

- найдем самый старший разряд, в котором цифра больше 1 — пусть это будет `pos`;
- чтобы итоговое число стало не меньше, какой-то из более старших разрядов надо увеличить, то есть сделать равным 1;

- а тогда во все разряды младше увеличенного можно поставить цифру 0, чтобы оно было минимальным из больших n;
- чтобы выбрать, какой разряд увеличивать до единицы, найдем первый нулевой разряд старше `pos` и поставим в него 1 (если такого разряда не нашлось, добавим новый).

Асимптотика времени работы ответа на запрос — $\mathcal{O}(\log_k n)$.

Задача B. Смешивание напитков

Автор задачи: Даниил Орешников, разработчик: Егор Юлин

Основной идеей для решения данной задачи будет наблюдение, что крепость кофе непрерывна. То есть если у нас изначально был кофе крепости P_0, а после какого-то изменения крепость стала равна P, то мы могли получить любую крепость от P_0 до P, выпив из тех же слоев, но меньше.

Далее будут использоваться следующие обозначения: P_0 — изначальная крепость кофе в стакане, t_i — необходимая крепость кофе в запросе. Кроме того, будем отдельно рассматривать случаи, когда $t_i < P_0$ и случаи, когда $t_i > P_0$. Заметим, что заменой всех $p_i \leftarrow -p_i$ и всех $t_i \leftarrow -t_i$ можно привести один случай к другому, поэтому далее опишем только решение для $t_i < P_0$, а обратный случай будет полностью аналогичен.

Для решения **первой подзадачи** можно было перебирать конкретное множество слоев, которые будут целиком или частично выпиты. Для каждого множества затем можно проверять, могла ли в какой-то момент крепость кофе быть равной t_i. Поскольку мы рассматриваем $t_i < P_0$, достаточно проверить, что если выпить эти слои целиком, итоговая крепость будет $\leqslant t_i$.

Итоговую крепость можно посчитать за $\mathcal{O}(n)$ или за $\mathcal{O}(1)$, если по ходу рекурсивного перебора множества слоев поддерживать суммарные количество кофеина и высоту. Среди всех множеств, которые подошли, надо выбрать то, у которого самый глубокий слой находится как можно выше. Итоговое время работы — $\mathcal{O}(q \cdot 2^n)$.

Для решения **второй подзадачи** можно соптимизировать перебор и перебирать, сколько первых слоев будут задействованы (то есть по сути перебирать ответ на запрос). Опять же, пользуясь непрерывностью, заметим, что достаточно проверить, что можно получить крепость $\leqslant t_i$.

А для такой проверки можно заметить, что если зафиксировать доступные верхние x слоев, достаточно выпить целиком те из них, у которых крепость больше t_i. Если этого не хватит, то, выпивая еще и слои с крепостью $\leqslant t_i$, итоговой крепости t_i мы не добьемся. Соответственно, перебирая x, выпьем все слои крепости $> t_i$, после чего проверим, что итоговая крепость стала $\leqslant t_i$. Итоговое время работы — $\mathcal{O}(q \cdot n^2)$.

В **третьей подзадаче** можно было соптимизировать предыдущее решение и вместо обычного перебора высот искать ответ бинарным поиском. Это позволяет делать монотонность по ответу: чем более высокую трубочку мы возьмем, тем шире будет интервал достижимых уровней крепости. Такое решение работает за $\mathcal{O}(q \cdot n \cdot \log n)$.

Это же решение оптимизируется до асимптотики $\mathcal{O}(q \cdot n)$ для **четвертой подзадачи**, если совместить идеи второй и третьей подзадач. А именно, что чтобы получить крепость $t_i < P_0$, нет смысла трогать слои с крепостью $\leqslant t_i$.

Будем перебирать для каждого запроса затрагиваемые слои кофе сверху вниз. Тогда для каждого очередного слоя j, если $p_j \leqslant t_i$, мы его пропускаем, а если $> t_i$, выпиваем целиком. Как только общая крепость напитка достигнет значения меньше t_i, мы нашли наш ответ. Для быстрого вычисления общей крепости напитка достаточно поддерживать суммы по еще не выпитым слоям, фигурирующие в числителе и знаменателе формулы для P.

Пятую подзадачу можно было решать с помощью бинарного поиска по ответу. Используя рассуждения, описанные выше, поймем, что достаточно выпивать все слои, крепость которых больше t_i, а если слои отсортированы по крепости, то интересующие нас слои всегда будут образовывать некоторый отрезок.

Таким образом, можно было предподсчитать префиксные суммы $p_i \cdot h_i$ и p_i по слоям, после чего для ответа на запрос

- делать бинпоиск по ответу;

- внутри бинпоиском находить интересующий нас отрезок слоев;
- выпивать их и проверять, что итоговый уровень крепости нам подходит.

Вместо внутреннего бинпоиска можно было отсортировать запросы по t_i и использовать метод двух указателей: для меньших t_i нам всегда будет нужна не меньшая высота трубочки, и на соответствующем суффиксе слоев нас будет интересовать не менее длинный отрезок. Время работы: $\mathcal{O}(q \cdot \log^2 n)$ или $\mathcal{O}(q \cdot \log n)$, причем решение с двумя указателями уже довольно близко к полному.

Шестая и седьмая подзадачи были расчитаны на некоторые частные решения. Условие на равенство высот всех слоев позволяет сократить высоты из формулы для P и просто считать его как среднее арифметическое всех p_i. Необходимость следить только за двумя значениями крепости позволяет легче проанализировать, что выгодно пить только из слоев крепости с определенной стороны от t_i по значению.

Для **восьмой подзадачи** необходима идея с двумя указателями из решения пятой. Пусть у нас есть две крепости t_i, t_j, при этом $t_i < t_j$, тогда ответ для крепкости t_i будет не меньше ответа для крепкости t_j.

Предподсчитаем ответы для всех $1 \leqslant t \leqslant P_0$ по убыванию и для всех $P_0 < t \leqslant 10^5$ по возрастанию симметрично. Дальше, как обычно, рассматриваем $t < P_0$. Пусть рассматриваемая сейчас крепость равна t, тогда

- будем перебирать выпиваемый слой сверху вниз;
- также будем поддерживать множество всех невыпитых слоев, которые остались выше, упорядоченное по p_i (например, с помощью `std::set`);
- как уже было отмечено выше, достаточно выпить все слои с крепостью $> t$, поэтому, встречая новый слой, добавим его в множество, а затем удалим из множества и выпьем все слои с $p_i > t$;
- если итоговая крепость оказывается $< t$, то мы нашли ответ для крепости t.

Предподсчитав ответ для всех значений крепости от 1 до 10^5, мы затем можем отвечать на запросы за $\mathcal{O}(1)$, получая время работы $\mathcal{O}(\max(p) \cdot \log n + q)$.

Для **полного решения** воспользуемся идеей предыдущей подзадачи, просто реализовав метод двух указателей не по всем возможным значениям крепости, а только по фигурирующим в запросах. То есть отсортируем все запросы по t_i и найдем ответ описанным алгоритмом, перебирая $t_i \leqslant P_0$ по убыванию и $t_i > P_0$ по возрастанию симметрично.

Задача C. Нечестная игра

Автор задачи и разработчик: Даниил Орешников

Заметим несколько важных критериев:

1. чтобы фишка могла находиться на удаляемой клетке $(i_{\text{cur}}, j_{\text{cur}})$, должна быть возможность за $\sum\limits_{t \leqslant \text{cur}} k_t$ шагов добраться из (r, c) до $(i_{\text{cur}}, j_{\text{cur}})$;
2. если по пути мы не оказываемся в клетке без неудаленных соседей, то каждый шаг меняет четность суммы номера строки и столбца, поэтому четность клетки $(i_{\text{cur}}, j_{\text{cur}})$ должна быть соответствующей;
3. если $\sum\limits_{t \leqslant \text{cur}} k_t$ строго больше длины какого-то пути из (r, c) в $(i_{\text{cur}}, j_{\text{cur}})$, и у конечной клетки есть хотя бы один сосед, то можно дойти до этой клетки по найденному пути, а после делать парные ходы в соседнюю клетку и обратно;
4. альтернативно — если в клетке $(i_{\text{cur}}, j_{\text{cur}})$ можно было оказаться ровно в тот момент, когда был удален последний ее сосед, то можно в ней находиться и в любой момент после, так как больше из нее перемещений не будет.

Таким образом, для одного из частичных решений **третьей подзадачи**, в которой компьютер выполняет случайные ходы, можно было находить расстояние между клетками как $|r-i_{\text{cur}}|+|c-j_{\text{cur}}|$ за исключением случаев, когда клетки имеют общую координату. Если ходы случайны, то с большой вероятностью в первые несколько ходов найдется клетка нужной четности, достижимая за пройденное расстояние, и при этом все оценки расстояний будут верны, так как количество удаленных клеток не позволяет «перекрыть» все кратчайшие маршруты между стартовой клеткой и конечной.

Первая подзадача позволяла не беспокоиться об отслеживании пройденного расстояния: все клетки становятся достижимыми в первый же ход. Если четность удаляемой клетки совпадает с четностью пройденного числа шагов (с учетом четности стартовой клетки), то можно сразу вывести, что фишка находилась на удаляемой клетке. Такое решение набирало неполный балл, потому что не учитывало клетки, в которых фишка могла оказаться «запертой». Больше баллов можно было получить, проверяя также при удалении клетки, что число ее неудаленных соседей равно нулю — в таком случае мы можем выиграть вне завивимости от ее четности.

Во **второй подзадаче** перемещения всегда происходят на один шаг. Из-за этого, во-первых, четность клетки меняется с каждым ходом, а во-вторых, легко пересчитывать, в каких клетках мы бы могли сейчас находиться. Достаточно поддерживать известные расстояния до каждой клетки и массив клеток, которые стали достижимы на последнем ходу. Тогда для нового хода достаточно перебрать их соседей и обновить тех из них, до которых расстоянее ранее не было известно. Оставшиеся детали решения совпадают с описанным выше. Аналогичные рассуждения также работали и в **пятой подзадаче**.

Четвертая подзадача выделялась тем, что любые две удаляемые подряд клетки были соседними по стороне. Поскольку во вводе перечислены все клетки, они образуют «змейку», покрывающую все поле. А в таком случае можно показать, что фишка не может оказаться «запертой», и с каждым ходом будет менять четность клетки, на которой она находится, в соответствии с четностью совершенных перемещений. Таким образом, эту подзадачу проходила версия полного решения, не учитывающая возможность фишки оказаться заблокированной в клетке без соседей.

В **шестой подзадаче** поле представляет собой полосу, поэтому достаточно просто отслеживать, на какие отрезки ее разбивают удаленные клетки, а также в каких отрезках и на клетках какой четности может находиться фишка. Если не придумать полное решение, то можно было набрать баллы за эту подзадачу, поддерживая эти отрезки с помощью декартова дерева, либо с помощью системы непересекающихся множеств, обрабатывая запросы с конца.

Полное решение сочетает в себе все описанные выше общие идеи. Только для поиска расстояний между клетками требуется делать поиск в ширину из стартовой клетки. Поскольку в момент, когда мы доходим до клеток с расстоянием $> \sum\limits_{t \leqslant \text{cur}} k_t$, мы еще не знаем, можно ли будет после следующего хода компьютера через них проходить, будем

- поддерживать глобальную очередь `bfs` с еще не обработанными клетками;
- на каждом ходу продолжать `bfs` с того состояния, на котором остановились в прошлом, при чем будем пропускать уже удаленные клетки;
- останавливать `bfs` при достижении им клеток на расстоянии больше пройденного;
- для всех клеток, соседних с удаляемой, проверять, сколько из их соседей еще не удалены — если все соседи удалены, и фишка может находиться в данной клетке в этот момент (расстояния хватает и совпадает четность), запомним эту клетку в отдельное множество.

Тогда ответом будет первая удаляемая клетка, для которой выполнятся описанные в начале этого разбора критерии. **Седьмая подзадача** позволяла не заметить, что на все решение можно сделать только один «общий» `bfs`, и запускать на каждый запрос новый `bfs` заново. **Восьмая подзадача** ограничивала стартовую клетку, что могло помочь пройти тесты решениям, некорректно обрабатывающим какие-то из описанных случаев или критериев.

Общее время работы полного решения — $\mathcal{O}(nm)$.

Задача D. Стековое кодирование

Автор задачи и разработчик: Даниил Орешников

Сразу отметим, что во всех подзадачах, в которых $\gamma = 1$, было возможно найти точный оптимальный ответ. В остальных подзадачах решением, укладывающимся в ограничения по времени и памяти, можно было найти некоторое приближение ответа.

Основная идея решения — динамическое программирование. Пусть $\texttt{dp}[j][i]$ — минимальное количество действий, необходимое, чтобы получить префикс массива a длины i, если последним действием `print` был выписан полуинтервал $[j, i)$. Переберем k — начало предыдущего выписанного отрезка. Тогда нам необходимо знать $\texttt{lcp}(k, j)$ — наибольшую общую последовательность, начинающуюся в позициях k и j.

Действительно, если до этого в стеке лежала последовательность элементов с полуинтервала $[k, j)$, а теперь лежит $[j, i)$, то надо сначала удалить из стека лишние элементы, а потом добавить не хватающие, оставив общие на месте. То есть на такое преобразование стека тратится $(j-k)-\texttt{common}$ удалений и $(i-j)-\texttt{common}$ добавлений, где $\texttt{common} = \min(j-k, i-j, \texttt{lcp}(k, j))$ — общий префикс этих двух отрезков.

Таким образом, надо обновить $\texttt{dp}[j][i]$ через $\texttt{dp}[k][j] + (i - k - 2 \cdot \texttt{common}_{k,j,i})$ по всем $k < j$. Если каждый раз считать `lcp` за линейное время, весь пересчет динамики будет работать за $\mathcal{O}(n^4)$, если же его предподсчитать за $\mathcal{O}(n^2)$, то динамика посчитается за $\mathcal{O}(n^3)$. Восстановить ответ можно стандартным образом: возвращаясь «назад» по динамике по одному отрезку за шаг.

Точные решения

В **первой и второй подзадачах** решение, отвечающее на запрос явным «извлечением» отрезка и подсчетом динамики для него с нуля, проходило на полный балл, потому что позволяло найти точное решение. Поскольку различных отрезков может быть не больше $\mathcal{O}(n^2)$, то эти две подзадачи проходились динамикой даже за $\mathcal{O}(n^4)$.

Уже соптимизированная до $\mathcal{O}(n^3)$ динамика позволяла также получить точный ответ в **третьей подзадаче**.

Для **подзадач с четвертой по седьмую** требовалось важное наблюдение, которое позволяло соптимизировать подсчет этой динамики до $\mathcal{O}(n^2)$. Вернемся к алгоритму пересчета динамики и поменяем порядок перебора индексов: будем перебирать j и k, а затем будем перебирать i. Теперь заметим, что обновления, которые мы делаем, выглядят следующим образом:

- для $i_0 = j + \min(j - k, \texttt{lcp}(k, j))$ мы обновляем $\texttt{dp}[j][i_0]$ через $\texttt{dp}[k][j] + ((j - k) - (i_0 - j))$;
- для $i < i_0$ с каждым следующим (в порядке уменьшения) i значение правой части увеличивается на 1, так как требует лишнего удаления из стека;
- для $i > i_0$ с каждым следующим (в порядке увеличения) i значение правой части увеличивается на 1, так как требует лишнего добавления в стек.

То есть на самом деле можно сначала для фиксированного j перебрать k и обновить только $\texttt{dp}[j][i_0]$, а затем, вне перебора k, перебрать i и обновить $\texttt{dp}[j][i]$ через $\min(\texttt{dp}[j][i+1], \texttt{dp}[j][i-1]) + 1$.

Это позволяет посчитать ту же самую динамику за $\mathcal{O}(n^2)$, что дает возможность

- в **в четвертой подзадаче** посчитать ее для всего массива a;
- в **пятой подзадаче** посчитать ее для каждого суффикса массива a (суммарно за $\mathcal{O}(n^3)$), после чего для ответа на запрос $[l, r)$ использовать $\texttt{dp}_l[\text{opt}][r - l]$;
- в **шестой и седьмой подзадачах** посчитать ее для всего массива a и вместо запросов $[l, r)$ отвечать на $[0, r - l)$, так как все отрезки равны.

При этом в **шестой и седьмой подзадачах** работает конструктивное решение. Перебрем максимальный размер стека за весь процесс кодирования m. Поскольку мы начинаем с пустого стека, то его длина будет принимать все значения от 0 до m, а значит за k операций вывода можно получить

любую длину массива от 0 до $k \cdot m$. Таким образом, можно для каждого m посчитать величину $m + \left\lceil \frac{r-l}{m} \right\rceil$ и выбрать среди них минимальную.

Неточные решения

Наивным решением можно было набрать определенное ненулевое количество баллов: для каждого отрезка $[l, r)$ просто $r - l$ раз сделаем `push` соответствующего элемента, после чего один раз сделаем `print`.

Можно было решать задачу **жадным алгоритмом**: в общем случае можно было посчитать z-функцию для каждого суффикса массива, после чего с ее помощью разными эвристиками стараться разбить отрезок запроса на как можно более похожие друг на друга. В подзадачах с $a_i = 1$ можно было стараться набрать длину стека, наиболее близкую к $\sqrt{r-l}$, потому что при $n = k^2$ оптимум достигается на $k \times \texttt{push}(1) + k \times \texttt{print}$.

Для решения **восьмой и девятой подзадач** можно было заметить, что оптимальный код выглядит следующим образом:

1. берется какой-то код для «левой части» (от левой границы до 2);
2. последний вывод заменяется на `push`(2), `print` и `pop`;
3. после берется оптимальный код для «правой части» с учетом текущего размера стека.

Одним из способов приблизить ответ был следующий алгоритм:

1. переберем длину стека в конце вывода левой части m в некоторой окрестности корня из ее длины;
2. возьмем соответствующее значение `dp`, предпосчитанного за $\mathcal{O}(n^2)$ на массиве из всех единиц;
3. добавим вывод числа 2 в конец;
4. для правой части найдем такое ближайшее к m число m_2, что длина правой части раскладывается в сумму слагаемых от m до m_2 включительно;
5. упорядочив слагаемые в этом разложении, получим в точности последовательность длин отрезков, выводимых командной `print` — останется только в нужных местах добавить команды `push` или `pop`.

Десятая и одиннадцатая подзадачи требовали подхода, основанного на корневой декомпозиции. Посчитать динамику на всех суффиксах массива не хватит времени и памяти. Но можно посчитать динамику на всех суффиксах, начинающихся с позиций, делящихся на некоторый s.

Иными словами, посчитаем $\texttt{dp}_x[j][i]$ — значения динамики для суффикса массива a, начинающегося в позиции $x \cdot s$. Теперь, для ответа на запрос, если $r - l < s$, посчитаем для него отдельную динамику за $\mathcal{O}(s^2)$, а иначе:

1. найдем ближайшее справа к l число m, делящееся на s;
2. для ответа в правой части возьмем $\texttt{dp}_{\frac{m}{s}}[\text{opt}][r - m]$;
3. для ответа в левой части посчитаем динамику с нуля на отрезке $[l, m)$ за $\mathcal{O}(s^2)$;
4. восстановим ответы и «объединим» их, преобразовав последнее состояние стека для левой части в первое для правой удалениями и добавлениями соответствующих элементов.

Такое решение работает за $\mathcal{O}\left(\frac{n^3}{s}\right)$ на предподсчет и $\mathcal{O}(s^2 + n)$ для ответа на запрос, что при $s \approx \sqrt{n}$ дает $\mathcal{O}\left(n^{\frac{5}{2}} + qn\right)$ времени работы и дает достаточно хорошее приближение ответа в большинстве случаев.

Задача E. Годовой отчет

Автор задачи и разработчик: Прохор Ларичев

Будем воспринимать кандидатов как вершины графа, а дружбу — как его ребра. Тогда требуется выбрать среди всех максимальных клик (полных подграфов) такую, в которой `xor` значений в вершинах максимален.

Существует алгоритм Брона-Кербоша, который позволяет решить эту задачу с достаточным запасом по времени, перебрав все максимальные клики в графе за время $\mathcal{O}(3^{\frac{n}{3}})$. Поскольку этот алгоритм, скорее всего, не известен большинству из участников, ограничения в задаче были расчитаны на другое полное решение, о котором будет написано ниже.

В **первой подзадаче**, как это часто бывает, решение полным перебором проходит по времени. Если за $\mathcal{O}(2^n)$ перебрать подмножество вершин, и затем за $\mathcal{O}(n^2)$ проверить, образуют ли они клику, можно выбрать среди них максимальную по описанным критериям.

В случае $k = 0$, как во **второй подзадаче**, `xor` любого множества вершин будет равен 0, поэтому остается просто найти максимальную клику в графе. Есть множество алгоритмов, позволяющих найти произвольную максимальную клику, но здесь достаточно было воспользоваться методом «Meet in the Middle», позволяющим сократить полный перебор.

Разобьем все вершины на произвольные две равные группыЖ для каждого подмножества первой группы за $\mathcal{O}(2^{\frac{n}{2}})$ определим, является ли оно кликой, а также найдем список общих соседей. Останется только для каждой клики первой группы посмотреть на множество их общих соседей и выбрать в нем подмножество, являющееся кликой второй группы. Этот предподсчет также можно сделать за $\mathcal{O}(2^{\frac{n}{2}} \cdot n)$.

В **третьей подзадаче** выполняется $k \leqslant 3$, то есть можно перебрать возможные значение `xor` ответа, которых не более восьми, и для каждого применить решение предыдущей подзадачи, просто искать во второй группе вершин не произвольную клику среди соседей клики первой группы, а клику с конкретным значением `xor` вершин. Такие клики можно предподсчитать за $\mathcal{O}(3^{\frac{n}{2}})$, а в таком случае для MitM выгодно делить вершины на группы немного разных размеров: первая больше, вторая меньше. Это и является главным шагом на пути к полному решению.

Частный случай **пятой подзадачи** требует найти независимо максимальную клику, содержащую первую вершину, и не содержащую, и если первая не меньше второй, то выбрать в ответ ее. Для **шестой подзадачи**, аналогично, требуется сравнить максимальные клики, содержищие четное или нечетное количество вершин из первой и второй половины, соответственно. Такие клики можно найти аккуратной динамикой по подмножествам, после чего решение аналогично MitM для второй подзадачи.

Решение **четвертой подзадачи** и **полное решение** требует обобщения решения третьей. Разобьем n вершин на группы размерами m_1 и m_2, и в первой полным перебором найти все клики и `xor` значений в них, а во второй — аналогично, найти все клики, но еще для каждой маски M построить битовый бор на `xor`'ах ее максимальных подклик.

Тогда, чтобы найти максимум, переберем маску клики M_1 в первой группе, возьмем множество общих соседей ее вершин из второй группы M_2, и в битовом боре для M_2 найдем за $\log \max(a)$ значение y, максимизирующее $\left(\bigoplus\limits_{v \in M_1} a_v\right) \oplus y$.

Битовые боры при предподсчете для второй группы можно строить с использованием Small-to-Large оптимизации (взять максимальный из боров подмасок без одного элемента и персистентно добавить все недостающие подклики), но поскольку размер каждого бора для маски без однй вершины может быть меньше половины всех значений, здесь эта оптимизация не сильно поможет. Достаточно было аккуратно реализовать построение этих боров внутри перебора подмасок за $\mathcal{O}(3^{m_2})$.

Суммарное время работы такого решения равно $\mathcal{O}((2^{m_1} + 3^{m_2}) \cdot \log \max(a))$, и остается только выбрать такое разбиение n на $m_1 + m_2$, при котором эта величина минимальна. Менее эффективная реализация решения проходила только четвертую подзадачу.