Problem A. Axis-Aligned Area

Идея: Георгий Корнеев
Разработка: Геннадий Короткевич

Единственная область на плоскости, которую можно ограничить с помощью четырёх отрезков, параллельных осям координат — это прямоугольник.

Если два отрезка образуют противоположные стороны прямоугольника, то настоящей длиной стороны прямоугольника сможет стать только меньший из этих двух отрезков (а у большего останется лишняя длина).

Значит, нужно разбить заданные четыре отрезка на две пары, чтобы произведение минимумов в этих парах (которое и окажется равно площади прямоугольника) было как можно больше.

Можно заметить, что всегда оптимально в одну пару отнести два наименьших отрезка (a_1 и a_2), а во вторую — два наибольших (a_3 и a_4). При этом минимумы в этих парах мы уже знаем: $a_1 \le a_2$ и $a_3 \le a_4$ по условию.

Следовательно, ответом на задачу является произведение $a_1 \cdot a_3$.

Problem B. Based Zeros

Идея: Николай Ведерников
Разработка: Геннадий Короткевич

Интуитивно кажется, что ответ на задачу должен быть *маленьким* — у большинства чисел в двоичной системе будет много нулей; а если в двоичной нулей мало, то наверняка в троичной их будет много; и так далее. Иными словами, кажется, что сложно построить тест, на котором нулей будет мало в любой системе счисления.

Попробуем формализовать эти рассуждения. Для начала напишем простое решение, которое будет перебирать системы счисления по возрастанию от 2 до n и считать в каждой из них число нулей. Важная оптимизация: как только цифр в числе стало не больше, чем было найдено нулей в какой-то из предыдущих систем счисления, можно остановить перебор. Назовём это решение *алгоритмом X*.

Решение в таком виде получает Time Limit Exceeded. Давайте попробуем понять, на каких тестах это решение может работать долго.

Переберём локально на своём компьютере все числа, у которых нулей в двоичной системе счисления не более 4. Таких чисел будет всего несколько миллионов. На каждом из них запустим алгоритм X.

Окажется, что среди таких чисел худшим случаем для нашего алгоритма будет число $n = 16\,760\,831$: оно имеет не более одного нуля в любой системе счисления, поэтому алгоритму X придётся перебирать все системы счисления до n включительно.

Посмотрим на числа, для которых максимальная система счисления в ответе превышает $2^{12} = 4096$. Таких чисел всего семь: помимо уже упомянутого 16 760 831, это числа 6143, 14 335, 262 079, 262 111, 524 031 и 524 285.

Для чисел же, которые имеют 5 и более нулей в двоичной системе, все ответы точно не превышают 4096, поскольку $4096^5 = 2^{60} > 10^{18}$.

Таким образом, получаем следующее решение:

- если на вводе одно из семи чисел, перечисленных выше, выводим найденный локально ответ;
- в противном случае, как мы уже показали, достаточно перебрать системы счисления до 4096 и вывести лучшие из них.

Кроме того, следующее простое улучшение к алгоритму X тоже позволяет получить Accepted: запоминать ответы на тестовые случаи, которые мы уже видели ранее в рамках того же теста, и не вычислять их заново. На тесте из 1000 чисел, равных 16 760 831, алгоритм X будет работать медленно, однако с тысячей худших для алгоритма X *различных* чисел такое решение может справиться.

Problem C. Colorful Village

Идея: Федор Царев, Геннадий Короткевич
Разработка: Геннадий Короткевич

В дереве из $2n$ вершин, вершины которого покрашены в n цветов, по две каждого цвета, нужно выбрать связное поддерево из n вершин, по одной каждого цвета.

Предположим, что мы знаем, что некоторую вершину r мы возьмём в искомое множество. Подвесим дерево за вершину r. Тогда связность нашего поддерева будет эквивалентна следующему условию: для каждой вершины v, входящей в множество, её родитель p_v тоже входит в множество.

Заведём булеву переменную b_v для каждой вершины дерева v. Значением $b_v = \texttt{true}$ будем обозначать, что вершина v входит в искомое множество, значением $b_v = \texttt{false}$ — что не входит. Тогда можно сформулировать следующие два условия, которые будут необходимы и достаточны.

- Из каждой пары вершин одного цвета, u и v, ровно одна входит в множество: $b_u \neq b_v$. Эквивалентная формула: $(b_u \vee b_v) \wedge (\neg b_u \vee \neg b_v)$.
- Если вершина v входит в множество, то её родитель p_v тоже в него входит: $(\neg b_v \vee b_{p_v})$.

Если объединить все эти условия, мы получим формулу в 2-CNF. Найти значения переменных, удовлетворяющие такую формулу, можно с помощью известного алгоритма за линейное время от её размера, что в нашем случае составит $O(n)$.

Единственный вопрос, который мы упустили — как найти вершину r. Есть как минимум два варианта.

- Взять обе вершины цвета 1 и попробовать каждую из них в качестве r. Если хотя бы в одном случае мы нашли решение, выводим его. В противном случае можно смело выводить -1, поскольку хотя бы одна из вершин цвета 1 точно должна входить в ответ.
- Найти любой центроид дерева и назначить его вершиной r. Действительно, по определению центроида, при его удалении дерево распадётся на компоненты связности размера не более n. Если все компоненты имеют размер строго меньше n, то центроид обязан входить в любой ответ. Если же есть одна компонента размера ровно n, не содержащая центроид, и эта компонента окажется корректным ответом (будет содержать каждый цвет по одному разу), то и остальная часть дерева, содержащая центроид и имеющая размер n, тоже окажется корректным ответом (тоже будет содержать каждый цвет по одному разу). Следовательно, если ответ существует, то существует и ответ, содержащий в себе центроид дерева.

Problem D. Divisibility Trick

Идея: Павел Маврин
Разработка: Павел Маврин

Есть множество способов решить эту задачу. Самое простое решение — сделать d копий числа d, и объединить их в одно длинное число: например, для числа $d = 12$ получим число 121212121212121212121212. Легко заметить, что это число будет делиться на d, и сумма его цифр также будет делиться на d. Длина такого числа не превышает 4000.

Другое возможное решение — построить граф, вершинами которого являются пары (остаток от деления числа на d, остаток от деления суммы цифр числа на d), и далее обходом в ширину найти кратчайший путь до вершины $(0, 0)$. Интересно, что с помощью такой идеи можно даже найти минимальное число, удовлетворяющее требованиям задачи.

Problem E. Every Queen

Идея: Павел Маврин

Если ферзь всего один, то можно вывести его позицию.

Если первый ферзь не бьёт второго, то существует всего $O(1)$ различных клеток, которые атакованы обоими ферзями. Для каждой из них за $O(n)$ проверим, является ли она ответом. Если ни одна из клеток не подошла, то решения не существует.

Если же первый ферзь атакует второго, то проведем прямую через них и найдем ферзя, который на ней не лежит. Если такого нет, то можно вывести позицию первого ферзя. Иначе существует $O(1)$ позиций, которые атакованы этими тремя ферзями. Каждую из них можно проверить за $O(n)$.

Как легко найти клетки, которые атакованы каким-то небольшим набором ферзей? Для каждого из ферзей переберём одну из четырёх прямых, которые через него проходят, а потом пересечём все прямые. Если все ферзи не находились на одной прямой, то таких клеток будет $O(1)$.

Problem F. First Solved, Last Coded

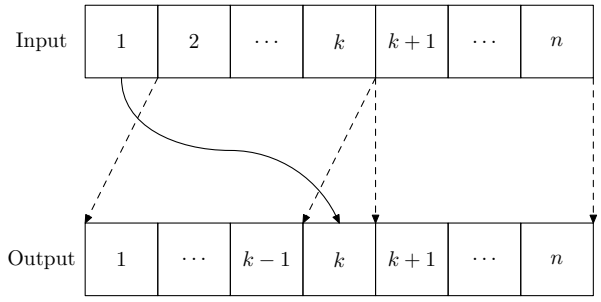
Идея: Артем Васильев
Разработка: Артем Васильев

Переформулируем задачу в более технических терминах: мы хотим преобразовать входной поток целых чисел в выходной, используя стек. На каждом шаге мы можем считать число из входного потока и положить его на вершину стека, либо достать число с вершины стека и вывести его в выходной поток.

Посмотрим, когда первое число, считанное из входного потока, переместится в выходной: пусть оно будет k-м выведенным в выходной поток. Поскольку это число было самым первым во входном потоке, оно всегда будет находиться на дне стека, и, когда мы переместим его в выходной поток, стек снова станет пустым. Так как стек стал пустым, то первые k чисел из входного потока перешли в первые k чисел в выходном.

Рассмотрим, что происходит до и после вывода этого числа:

- Числа на позициях $2 \dots k$ из входного потока перейдут в позиции $1 \dots k-1$ в выходном потоке;
- После этого, числа с позиций $k+1 \dots n$ из входа окажутся на позициях $k+1 \dots n$ в выходе.



Обе эти ситуации можно рассматривать как подзадачи вида "Можно ли преобразовать подотрезок $\texttt{A}[i..i+l]$ в подотрезок $\texttt{B}[j..j+l]$, начав (и закончив) с пустым стеком?". Такая формулировка напрямую ведет к решению с помощью динамического программирования: обозначим это утверждение как $\texttt{can}_{i,j,l}$. Будем вычислять все $\texttt{can}_{i,j,l}$ в порядке возрастания l. Для вычисления $\texttt{can}_{i,j,l}$ рассуждаем так же, как и выше: пусть $\texttt{A}[i]$ будет выведено в позиции $j+k$. Чтобы это было возможно, необходимо, чтобы выполнялись следующие условия:

- $\texttt{A}[i]$ должно быть равно $\texttt{B}[j+k]$,
- Из $\texttt{A}[i+1..i+k]$ нужно уметь получать $\texttt{B}[j..j+k-1]$: мы вычислили это на одном из предыдущих шагов как $\texttt{can}_{i+1,j,k-1}$;

- Из $\texttt{A}[i+k+1..i+l]$ нужно уметь получать $\texttt{B}[j+k+1..j+l]$: мы вычислили это как $\texttt{can}_{i+k+1,j+k+1,l-k-1}$

Из этих условий можно получить решение с помощью ДП за $O(n^4)$: для каждого из $O(n^3)$ состояний (i, j, l) перебираем, чему будет соответствовать $\texttt{A}[i]$ и проверяем условия выше. Восстановление ответа реализуется стандартной техникой: запомним, для какого k мы выставили значение $\texttt{can}_{i,j,l}$ в `true`, и рекурсивно восстановим ответ, начиная с $\texttt{can}_{1,1,n-1}$.

Problem G. Game of Nim

Идея: Федор Царев
Разработка: Федор Царев

Задача может быть решена с помощью динамического программирования.

Пусть $a_{i,j,k}$ обозначает число способов разделения i камней на несколько кучек таким образом, чтобы размер самой большой из них был равен j, а результат операции XOR (исключающее ИЛИ) всех размеров кучек равен k. Диапазон для i составляет от 0 до $n-p$, для j — от 0 до $n-p$, для k — от 0 до $B-1$, где B — это наименьшая степень двойки, которая больше или равна максимуму $(n-p,\ p)$.

Значения $a_{i,j,k}$ могут быть вычислены за время $O((n-p)^4)$: $a_{i,j,k}$ представляет собой сумму по всем p от 0 до j (мы перебираем размеры второй по величине кучки в наборе) $a_{i-j,p,k\oplus j}$ (мы также используем то, что операция XOR является обратной себе).

Алгоритм может быть ускорен с использованием дополнительного массива: $b_{i,j,k} = a_{i,0,k} + a_{i,1,k} + \ldots + a_{i,j,k}$. Используя этот массив, $a_{i,j,k}$ может быть вычислено как $b_{i-j,j,j\oplus k}$. Значения $b_{i,j,k}$ могут быть вычислены за время $O((n-p)^3)$, так как $b_{i,j,k} = b_{i,j-1,k} + a_{i,j,k}$. Таким образом, общее время работы составит $O((n-p)^3)$.

Problem H. H-Shaped Figures

Идея: Геннадий Короткевич
Разработка: Геннадий Короткевич

Для начала выкинем из множества отрезки, параллельные PQ. Теперь отрезки, содержащие точку P, отнесём в множество A, а отрезки, содержащие точку Q — в множество B.

Любая пара отрезков, один из A и один из B, подходит тогда и только тогда, когда они не пересекаются. Ответом на задачу является $|A| \cdot |B| - c$, где c — число пар пересекающихся отрезков.

Заметим, что пересекаться два отрезка могут либо с одной стороны от прямой PQ, либо с другой, но не одновременно. Поэтому решим задачу с двух сторон от прямой PQ независимо и сложим ответы.

При решении задачи с одной из сторон можно разрезать все отрезки по точкам P и Q и только использовать те части, которые ведут в нужную полуплоскость.

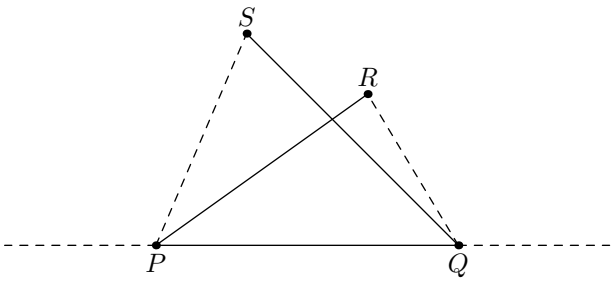
Свели задачу к следующей:

- Рассмотрим полуплоскость слева от вектора $\overrightarrow{PQ}$. Дано множество отрезков A, ведущих из точки P в эту полуплоскость, а также множество отрезков B, ведущих из точки Q в эту же полуплоскость. Найти число пар пересекающихся отрезков.

Рассмотрим два отрезка: отрезок PR из множества A и отрезок QS из множества B. Эти два отрезка пересекаются тогда и только тогда, когда оба следующих утверждения верны:

- векторное произведение $\overrightarrow{PR} \times \overrightarrow{PS}$ неотрицательно;
- векторное произведение $\overrightarrow{QR} \times \overrightarrow{QS}$ неотрицательно.

Иными словами, если точка S находится слева (или на той же прямой) от точки R и относительно точки P, и относительно точки Q, то отрезки пересекаются.



Отсортируем концы всех отрезков по углу относительно точки P с помощью векторного произведения в компараторе и выдадим каждой точке целое число p_i, обозначающее её номер в порядке возрастания против часовой стрелки от вектора $\overrightarrow{PQ}$ (точкам на одной прямой с точкой P выдадим одинаковые номера). Аналогично, отсортируем концы всех отрезков по углу относительно точки Q и выдадим каждой точке значение q_i.

Теперь нужно найти число пар точек — точка R из множества A и точка S из множества B — таких, что $p_R \le p_S$ и $q_R \le q_S$.

Число таких пар можно найти с помощью алгоритма сканирующей прямой с деревом отрезков или деревом Фенвика за $O(n \log n)$.

Problem I. Intersegment Activation

Идея: Артем Васильев
Разработка: Артем Васильев

Начнем с того, что придумаем стратегию, которая открывает клетку с номером 1. Ее покрывают n барьеров: $(1, 1)$, $(1, 2)$, ..., $(1, n)$. Есть 2^n разных состояний этих n барьеров, и ровно одно из них (то, в котором все барьеры отключены) открывает клетку. Попробуем посетить все эти 2^n состояний. Для этого идеально подойдет код Грея: он перечисляет все двоичные последовательности длины n таким образом, что две соседние отличаются изменением одного бита.

Приведем конструкцию такого кода. Начиная со неизвестного стартового состояния, сделаем $2^n - 1$ шагов:

- На нечетных шагах 1, 3, ..., $2^n - 1$, сделаем запрос $(1, 1)$;
- На шагах 2, 6, 10, ... (делящихся на 2, но не на 4), запрос $(1, 2)$;
- ...
- На шагах $2^i, 3 \cdot 2^i, 5 \cdot 2^i, \ldots$, запрос $(1, i + 1)$;
- ...
- На шаге с номером 2^{n-1}, запрос $(1, n)$.

Эта последовательность запросов дает стандартный двоичный зеркальный код Грея, выраженный в терминах того, какой бит изменяется на каждом шаге. Сделав такую последовательность запросов, мы посетим каждое состояние ровно однажды.

Осталось понять, в какой момент клетка 1 стала открытой. Заметим, что на каждом втором шаге мы делаем запрос $(1, 1)$; рассмотрим, что происходит только после таких запросов. Есть три варианта того, как может измениться k:

- $k \to k + 1$: последний запрос открыл клетку 1;

- $k \to k-1$: последний запрос закрыл клетку 1, сделаем запрос $(1,1)$, чтобы открыть клетку обратно;
- k не изменилось: состояние клетки 1 не поменялось.

После того, как мы открыли первую клетку, повторим тот же процесс для следующей клетки. В этот раз, мы работаем с $n-1$ барьером, которые ее покрывают, так что алгоритм, описанный выше, использует 2^{n-1} запросов. Таким образом, мы откроем все клетки за не более чем $2^n + 2^{n-1} + \ldots + 2^1$ запросов, что не превосходит $2^{n+1} \le 2048 \le 2500$.

Problem J. Jumping Frogs

Идея: Павел Маврин
Разработка: Павел Кунявский

Сначала нужно понять, как устроен ответ для фиксированного c. Пусть у нас есть какой-то корректный ответ. И пусть в нем есть две лягушки такие, что $i < j$, а значит, $a_i < a_j$, и при этом i прыгнула налево, а j — прыгнула направо. Заметим, что если поменять их местами в конечном ответе, он все еще останется корректным, так как расположенная правее лягушка тем более могла прыгнуть налево в ту же точку, и наоборот. Таким образом, если есть какой-нибудь ответ, то есть ответ, в котором c самых правых лягушек прыгали налево, а остальные — направо.

Значит, чтобы ответ для данного c существовал, нужно, чтобы c самых правых лягушек могли перепрыгнуть налево на c самых левых конечных позиций, и наоборот — $n-c$ самых левых лягушек могли перепрыгнуть направо на $n-c$ самых правых конечных позиций. Будем определять списки c, для которых можно сделать первое и второе независимо, а в конце пересечем эти два списка. Задачи полностью симметричны и решаются одинаково, поэтому дальше будем обсуждать первую из них.

Для фиксированного c задача решается жадно. Самая правая лягушка должна прыгать на c-ю позицию, следующая с конца — на $c-1$-ю, и так далее (т.к. если это не так, можно поменять двух лягушек местами, и станет не хуже). Что так можно сделать, можно проверить за линейное время. Это дает решение за $O(n^2)$, что слишком медленно для этой задачи.

Последнее наблюдение, которое осталось сделать — это заметить, что если для некоторого c ответа не существует, то и для всех больших его тоже не существует. И действительно, чтобы ответ существовал для большего c, этим c лягушкам придется переместиться на набор позиций, все из которых правее, а уже на эти не получилось. Таким образом, существует некоторое ограничение C, что для всех $c \le C$ ответ существует, а для больших нет. Это C можно найти с помощью бинарного поиска, в таком случае время работы составит $O(n \log n)$.

Более того, из решения выше следует, что ответ на конечную задачу —это всегда отрезок чисел. Но для решения это замечать необязательно.

Problem K. Kitchen Timer

Идея: Андрей Станкевич
Разработка: Андрей Станкевич

У этой задачи есть два возможных правильных подхода.

Решение 1

Мы хотим получить равенство $x = a_1 + a_2 + \ldots + a_k$, где каждое a_i равно $2^{p_i} - 1$, и k минимально возможное. Таким образом,

$$x = 2^{p_1} - 1 + 2^{p_2} - 1 + \ldots + 2^{p_k} - 1 = 2^{p_1} + 2^{p_2} + \ldots + 2^{p_k} - k,$$

$$x + k = 2^{p_1} - 1 + 2^{p_2} - 1 + \ldots + 2^{p_k} - 1 = 2^{p_1} + 2^{p_2} + \ldots + 2^{p_k}.$$

Следовательно, нам нужно найти такое минимальное k, что $x + k$ может быть представлено в виде суммы k степеней двойки.

Минимальное количество степеней двойки, которые нужно сложить, чтобы получить целое число v, равно количеству единиц в двоичной записи v. Любое большее количество степеней двойки также может быть использовано, но, разумеется, не более чем v, потому что мы всегда можем разбить $2^p = 2^{p-1} + 2^{p-1}$.

Таким образом, мы перебираем k, чтобы найти минимальное k, такое что $x + k$ содержит не более k единиц в своей двоичной записи. Ответ равен $k - 1$. Это решение выполняется за $O(\log^2 x)$, что достаточно быстро.

Решение 2

Жадный подход также работает для этой задачи. Найдем максимальное возможное p, такое что $2^p - 1 \le x$. Вычтем $2^p - 1$ из x и, если $x > 0$, повторим. Ответом будет количество итераций, минус один.

Для тех, кто любит математику, докажем жадное решение. Пусть снова

$$x = 2^{p_1} - 1 + 2^{p_2} - 1 + \ldots + 2^{p_k} - 1,$$

и $p_1 \ge p_2 \ge \ldots \ge p_k$.

Наблюдение. Если $p_{i-1} = p_i$ и $i < k$, мы можем переписать

$$2^{p_{i-1}} - 1 + 2^{p_i} - 1 + 2^{p_{i+1}} - 1 = 2^{p_{i-1}+1} - 1 - 1 + 2^{p_{i+1}} - 1 = 2^{p_{i-1}+1} - 1 + 2^{p_{i+1}-1} - 1 + 2^{p_{i+1}-1} - 1.$$

Если $p_{i+1} = 1$, мы можем опустить $2^{p_{i+1}-1} - 1 = 0$ и уменьшить количество слагаемых.

Повторяя шаг из Наблюдения, пока это возможно, мы приходим к выводу, что существует оптимальное решение, в котором нет равных степеней двойки в слагаемых, за исключением, возможно, двух минимальных слагаемых. Рассмотрим такое решение, в котором $p_1 > p_2 > p_3 > \ldots > p_{k-1} \ge p_k$.

$$x = 2^{p_1} - 1 + 2^{p_2} - 1 + \ldots + 2^{p_{k-1}} - 1 + 2^{p_k} - 1 \le (2^{p_1} - 1) + \left(2^{p_1-1} - 1 + 2^{p_1-2} - 1 + \ldots + 2^{p_k} - 1\right) + 2^{p_k} - 1 =$$

(мы добавили все пропущенные слагаемые и добавили еще $2^{p_k} - 1$, если $p_{k-1} > p_k$)

$$= (2^{p_1} - 1) + \left(2^{p_1-1} + 2^{p_1-2} + \ldots + 2^{p_k} + (2^{p_k} - 1)\right) - (p_1 - p_k) =$$

$$= (2^{p_1} - 1) + (2^{p_1} - 1) - (p_1 - p_k) \le (2^{p_1} - 1) + (2^{p_1} - 1) < 2^{p_1+1} - 1,$$

поэтому жадное решение правильно находит ведущее слагаемое.

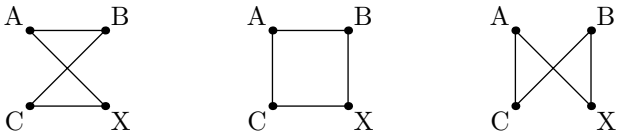
Problem L. Loops

Идея:	Павел Маврин
Разработка:	Павел Маврин

Идея решения: пусть у некоторого квадрата зафиксирован порядок некоторых трех элементов. Тогда, добавив четвертый элемент в нужное место этого порядка, можно получить любой из трех типов циклов.

Например, пусть поставлены элементы A, B и C так, что $A < B < C$. Тогда можно добавить четвертый элемент X в этот порядок следующим образом:

$A < B < C < X$ $A < B < X < C$ $A < X < B < C$



Используя эту процедуру, можно последовательно добавлять элементы таблицы сверху вниз, слева направо. Первую строку и первый столбец выставим как угодно, далее каждый раз добавляем элемент в текущий порядок так, чтобы получился цикл требуемого типа.

Для реализации этого решения нам нужна структура данных, которая хранит элементы в некотором порядке, и позволяет делать две операции:

1. вставлять новые элементы в середину;
2. сравнивать положения двух элементов.

Для этого подойдет, например, дерево поиска по неявному ключу, которое умеет делать обе эти операции за время $O(\log n)$. Таким образом, суммарное время работы будет $O(nm \log(nm))$.

Другой вариант — заметить, что нам нужно сравнивать только элементы одной строки. Можно это делать за $O(m)$, суммарное время работы будет $O(nm^2)$, что также достаточно для того, чтобы уложиться в ограничение по времени.

Также отметим, что можно решить задачу за время $O(nm)$, без применения дерева поиска по неявному ключу. Для этого нужно заметить, что нам важен порядок только соседних элементов в таблице. Можно следить за его изменением при добавлении новых элементов, тогда мы сможем понять, куда добавить новый элемент, за $O(1)$. Чтобы построить ответ, можно поддерживать связный список элементов, или же построить ориентированный граф, соответствующий требуемому порядку, и в конце построить его топологическую сортировку.

Problem M. Missing Vowels

Идея: Дмитрий Штукенберг
Разработка: Дмитрий Штукенберг, Геннадий Короткевич

В задаче требуется дополнить строку s гласными до строки f, если это возможно. Условие немного напоминает классические задачи на динамическое программирование, но в данной задаче достаточно жадного алгоритма. А именно, пройдём параллельно по строкам f и s, и по каждой паре текущих символов $\langle f_i, s_j \rangle$ выберем один из трёх вариантов поведения (конец строки для простоты также принят за особый согласный символ):

1. сдвигаемся вперёд в обеих строках, если $f_i = s_j$; если одновременно был достигнут конец обеих строк, то останавливаемся с ответом “Same”;
2. сдвигаемся вперёд в строке f, если f_i — гласный и $f_i \neq s_j$;
3. останавливаемся с ответом “Different”, если f_i — согласный (включая символ конца строки) и $f_i \neq s_j$.

Почему жадный алгоритм работает? Примем инвариант: на каждом шаге сопоставления подстрока $s_{1\ldots j}$ может быть дополнена до подстроки $f_{1\ldots i}$. Параллельное сопоставление почти во всех случаях при продвижении по строке f предполагает единственный вариант продвижения по строке s, сохраняющий инвариант, за исключением случая совпадения гласных символов ($f_i = s_j$).

При $f_j = s_i$ возможно как (а) продвижение по строке s, так и (б) добавление копии f_i перед s_j, без продвижения по s. Однако, вариант (б) в смысле сопоставления строк беднее: он накладывает дополнительное ограничение (необходимо найти место для s_j в строке f после символа f_i), не давая возможностей (ведь вставить ещё одну копию гласного s_j можно в любой момент). Поэтому вариант (а) всегда предпочтительнее.