Задача 1. Банковские карты

Задачу можно переформулировать так: найти минимальное натуральное число N, которое нельзя получить из заданного натурального числа X вычеркиванием некоторых цифр.

Основным понятием, помогающим нам понять решение этой задачи, будет понятие блока. *Блок* — это отрезок цифр числа X, содержащий все цифры от 0 до 9. Ясно, скажем, что если число X можно разбить на 5 блоков, то из него можно вычеркиванием цифр получить любое пятизначное число. Оказывается, с небольшими оговорками верно и обратное.

Напомним, что *суффиксом* строки называется её отрезок с некоторого места и до конца. Например, суффиксами строки 12345 являются строки 12345, 2345, 345, 45 и 5.

Итак, выделим в числе X минимальный (самый короткий) блок-суффикс. То есть возьмём несколько (как можно меньше) последних цифр X, образующих блок. Например, если $X = 12013456789$, то в этот блок войдут цифры 2013456789, а если $X = 112233445566778899001122334455667788990 0$, то в блок войдут цифры 1223344556677889900. Повторим эту операцию для оставшихся цифр X, потом ещё раз, и так далее, пока это возможно. Число X окажется представлено в виде $S_0B_1B_2B_3\ldots B_L$, где B_i — блоки, а S_0 — *неполный блок*, то есть некоторый (возможно, пустой) отрезок числа X, содержащий не все цифры от 0 до 9. Кроме того, по построению блоков B_i они удовлетворяют следующему условию: никакой суффикс блока B_i (не совпадающий со всем B_i) не является блоком, то есть все суффиксы B_i являются неполными блоками.

Теперь перейдём к решению нашей задачи. Пусть d_0 — это минимальная из цифр от 1 до 9, не встречающаяся в S_0. Пусть S_1 — это суффикс блока B_1 после первого вхождения в него цифры d_0. Пусть d_1 — это минимальная из цифр от 0 до 9, не встречающаяся в S_1. И так далее, последовательно строим S_2, d_2, …, S_L, d_L. Тогда ответом является число $N = d_0d_1\ldots d_L$.

Приведём пример: пусть $X = 5129312564339071801234567 89$. Тогда $B_2 = 0123456789$, $B_1 = 25643390718$, $S_0 = 512931$. Получаем $d_0 = 4$, значит $S_1 = 3390718$, значит $d_1 = 2$, значит $S_2 = 3456789$, значит $d_2 = 0$. Ответ $N = 420$.

Почему же этот метод всегда работает верно? Во-первых, на каждом шаге очередное d_i можно выбрать, так как S_i — это неполный блок (для S_0 это верно по построению, а S_i — это суффикс B_i, а значит, неполный блок). Во-вторых, на каждом шаге очередное S_i можно выбрать, так как B_i — это блок. Тем самым, метод всегда завершает работу. Осталось проверить, что найденный им ответ — правильный.

Проверим, что число N нельзя получить из числа X. Действительно, первая цифра числа N (d_0) отсутствует в S_0, значит её придётся брать из B_1 или ещё правее. Но тогда вторую цифру числа N (d_1) придётся брать из B_2 или правее, и так далее, и цифру d_L брать неоткуда.

Теперь докажем, что любое меньшее число можно получить. Пусть $M < N$. Если в M меньше цифр, чем в N, то его можно получить, взяв первую цифру из B_1, вторую из B_2, и так далее. Пусть в M столько же цифр, сколько в N: $M = e_0e_1\ldots e_L$. Пусть k — номер первой цифры, где M и N различаются: $e_0 = d_0$, $e_1 = d_1$, …, $e_{k-1} = d_{k-1}$. Тогда из того, что $M < N$, получаем $e_k < d_k$. Выберем цифру e_0 из B_1, цифру e_1 из B_2, и так далее, цифру e_{k-1} из B_k. Тогда по построению цифры d_k все меньшие цифры можно найти в B_k правее цифры d_{k-1}, а значит мы сможем взять цифру e_k из B_k. Далее берём цифру e_{k+1} из B_{k+1} и так далее, и получаем число M.

В решении остался неразобранным один случай: когда в S_0 встречаются все цифры от 1 до 9 (но не встречается цифра 0, иначе этот блок был бы полон). Оказывается, тогда необходимо взять $d_0 = 0$, но так как число не может начинаться с 0, необходимо дописать к нему слева цифру 1. Например, если $X = S_0 = 123456789$, то получаем $N = 10$. Доказательство правильности этого приёма оставим в качестве упражнения.

Время работы этого решения линейно по длине X. Программа получается довольно простая, и не требует знания каких-либо сложных алгоритмов или конструкций языка.

В этой задаче возможны и другие решения, например, можно применить бинарный поиск по ответу и научиться проверять, все ли числа, меньшие N, можно получить из X, однако реализовать их гораздо сложнее, и они имеют большее (хотя и удовлетворяющее ограничениям в условии задачи) время работы.

Задача 2. Файловый менеджер

Общий шаг алгоритма – k раз находить кратчайшие пути из одной вершины в другую на предварительно построенном графе.

Будем называть имена файлов строками.

Рассмотрим полный ориентированный граф с N вершинами, где вершина i соответствует файлу номер i. Вес ребра (i, j) – минимальная стоимость добраться из позиции курсора номер i в позицию j, используя только одну возможность менеджера (нажать клавишу вверх, нажать клавишу вниз, нажать клавишу Alt и последовательность символов). Ребра, реализующие первые две возможности (up, *d*own), соединяют все вершины, номера которых отличаются на 1. Вес таких ребер равен 1.

Самая главная сложность этой задачи состоит в том, чтобы быстро найти веса ребер для последней возможности (Alt + <Text>). Вес такого ребра (i, j) – минимальная длина последовательности нажатий клавиш, которая переводит курсор с i-го файла на j-й. Если из i в j нельзя попасть только с помощью "Alt", то вес ребра будет плюс бесконечность.

Таким образом, чтобы найти последовательность нажатий минимальной длины, перемещающую курсор из позиции A в позицию B, надо найти кратчайший путь из A в B в нашем графе.

Если запустить алгоритм Дейкстры из вершины A, то в $d[B]$ получим искомую длину.

Пусть мы знаем, как быстро вычислять массив $common[i][j]$ = максимальный общий префикс строк номер i и j. Тогда $d[i][j]$ – вес ребра (i, j) – равен максимуму среди $(common[t][j] + 2)$ $i <= t < j$ (если $i > j$, то t пробегает значения i, $i + 1$, …N, 1, …, $j - 1$). Это верно, так как надо нажать сначала Alt, потом $max(common[t][j])$ букв, потом еще одну. Если нажать менее чем $d[i][j]$ клавиш, то курсор попадет на какую-нибудь строку между i и j. Если $d[i][j]$ получилось больше, чем длина j-го слова + 1, то из i в j нельзя попасть только с помощью "Alt", $d[i][j]$ присваиваем плюс бесконечность.

Например, для набора

```
aaa
baa
aa
bbbb
bbcc
```

$d[1][5] = 4$, $d[1][3] = +\infty$.

Вычисление D по $common$ несложно реализовать за $O(N^2)$ следующим образом:

$d[j - 1][j] = common[j - 1][j] + 2$,
$d[j - 2][j] = max(common[j - 2][j] + 2, d[j - 1][j])$,
…
$d[i][j] = max(common[i][j] + 2, d[i + 1][j])$.

Обсудим три способа найти $common[i][j]$ для всех i, j.

а) $O(N*\log(N)*L)$, где L – максимальная длина слов.

Отсортируем все имена в лексикографическом порядке, пусть строка i имеет номер $P(i)$ в отсортированном списке.

Найдем $c[P(i)]$ – размер наибольшего общего префикса строк $P(i)$, $P(i + 1)$. Тогда $common[i][j]$, где $P(i) < P(j)$ – минимум среди всех $c[t]$, $P(i) \le t < P(j)$. Действительно, если $P(i)$ и $P(j)$ имеют общий префикс длиной L, то $c[t] \ge L$, причем должно быть хотя бы одно равенство.

Тогда
$common[P(j - 1)][P(j)] = c[P(j - 1)]$,
$common[P(j - 2)][P(j)] = max(c[j - 2], common[P(j - 1)][P(j)])$,
...
$common[P(i)][P(j)] = max(c[P(i)], common[P(i + 1)][P(j)])$.

б) $O(N^2 + N*L)$ (предложено одним из участников)

Будем искать $common$ методом динамического программирования.

Пусть $B[i + 1][j]$ – такое $1 \le t \le j$, что $common[i + 1][t]$ максимально.

Допустим, вычислены все $common[a][b]$ и $B[a][b]$, $a \le i$, $b \le i$. Научимся находить $common[i + 1][j + 1]$ и $B[i + 1][j + 1]$ при условии, что для всех пар $(i + 1, t)$, $t \le j$, все уже вычислено. Возьмем $A = B[i + 1][j]$. Тогда первые $min(common[i + 1][A], common[A][j + 1])$ символов у строк $(i + 1)$ и $(j + 1)$ совпадают. Тогда если

$common[j + 1][A] < common[i + 1][A]$, то

$common[i + 1][j + 1] = common[j + 1][A]$ и

$B[i + 1][j + 1] = A$.

В противном случае находим $common[i + 1][j + 1]$ непосредственно сравнивая символы с номерами $common[i + 1][A] + 1, \ldots, L$ у строк $(i + 1)$, $(j + 1)$, пока не найдется несовпадения. При этом

$B[i + 1][j + 1] = max(common[i + 1][j + 1], B[i + 1][j])$.

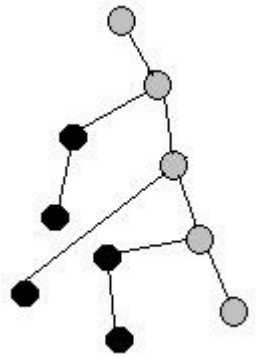
Таким образом, для каждого $(i + 1)$ непосредственных сравнений будет не более L, остальных действий $O(N)$, так что заявленная оценка верна.

в) $O(S + N^2)$, где S – суммарная длина всех имен файлов.

Допишем к концу каждого имени файла символ “$”. Построим сжатый бор из таких расширенных строк (подробнее о борах см. **[1]**). Тогда каждой строке будет соответствовать лист в боре (у которого путевая метка равна расширенной строке). В противном случае получим, что префиксом какой-то строки A будет S\$, где S – строка без листа, тогда $A = S$, чего не может быть по условию.

Тогда $common[i][j]$ - длина путевой метки наименьшего общего предка для пары листьев, соответствующих строкам i, j (обозначается $LCA(i, j)$).

Вспомним, как алгоритм поиска в глубину раскрашивает вершины в процессе работы: вначале все вершины белые; когда вершина только начинает обрабатываться, ее красят в серый цвет; после того, как все ее сыновья (или вершины, в которые из нее ведут ребра) станут черными (обработанными), она становится черной.



Представим себе, что обходим в глубину из корня произвольное дерево.

Предложение. Пусть вершину j только что перекрасили в серый цвет (это значит, что все ее сыновья еще белые). Тогда $LCA(i, j)$ для любой черной i будет $q(i)$ - наиболее удаленная от корня серая вершиной, являющейся предком i.

Так как конфигурация серых и черных вершин меняется по ходу выполнения алгоритма, то будем различать $q(i)$ для разных моментов времени (в частности, когда i не является черной, $q(i)$ не определена).

Итак, будем обходить в глубину из корня наш бор как дерево. В массиве $q[i]$ в любой момент времени будет храниться $q(A)$ в текущий момент времени, где A – лист, соответствующий строке i. Также будет храниться список листьев, уже ставших черными. Тогда при перекрашивании очередного листа в серый цвет надо пробежаться по списку уже черных и проставить соответствующие значения $common$. При перекрашивании листа в черный цвет надо добавить его в список черных. Если при обработке любой вершины u оказывается, что $q[i]$ для какого-то i ссылается на черную вершину, то $q[i]$ – значение $q(A)$ для предыдущего момента времени, и $q[i]$ надо изменить на u.

Подробнее об алгоритме поиска LCA см. [2] среди задач к разделу «Обход в глубину».

Литература.

[1]. Дэн Гасфилд. «Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология»

[2]. Кормен, Лейзерсон, Ривест. «Алгоритмы: построение и анализ»

Задача 3 Приключение

Назовем рослостью человека сумму его роста по плечи плюс длина его рук. Таким образом, рослость — максимальная глубина ямы, из которой человек может выбраться самостоятельно.

Вначале решим более простую задачу: могут ли все люди выбраться из ямы? Если да, то кто-то из них выбирается последним, таким образом, его рослости достаточно чтобы выбраться из ямы самостоятельно. Таким образом, последним может выбираться человек с максимальной рослостью. При этом, этот человек может помогать другим, встав в самый низ колонны из людей, подняв ее на высоту своего роста. Таким образом, если все люди выбираются из ямы, то они могут выбираться в порядке увеличения рослости (самый рослый — в основании). Для этого им потребуется создать колонну, в основании которой стоит самый рослый человек, на нем следующий по рослости и так далее.

Вернемся к решению исходной задачи.

Пусть нам известно, какие люди могут выбраться из ямы, а какие — нет. Тогда мы можем образовать из всех людей колонну, причем те люди, которые не смогут выбраться стоят в основании колонны, поднимая остальных на сумму своих высот. При этом люди, которые выбираются из ямы стоят по увеличению рослости, если смотреть сверху вниз.

Построим людей в колонну по увеличению рослости (самый рослый — внизу) и занумеруем их сверху вниз. Пусть среди первых i человек из ямы могут выбраться j, при условии, что все остальные люди им помогают. Переместим оставшихся $(i - j)$ человек в основание колонны, при этом она поднимется на сумму их ростов. Таким, образом, нам выгодно, чтобы суммарный рост выбравшихся был минимален, так как при этом суммарный рост людей, переставленных в основание колонны, максимален.

Обозначим как P исходную высоту колонны (сумму ростов всех людей), а через $p_{i,j}$ — максимально возможную высоту оставшейся части колонны, если мы рассмотрели i человек, и из ямы выбираются j из них. Будем пересчитывать $p_{i,j}$ при помощи динамического программирования. В начале проинициализируем $p_{0,0} = P$, а остальные $p_{0,j}$ — значением «минус бесконечность».

Вычислим $p_{i,j}$. Существуют два варианта:

Человек с номером i выбрался из ямы. Для этого должно выполняться $p_{i-1,j-1} + l_i \geq H$ (рост самого человека в формулу не входит, так как он уже включен в $p_{i-1,j-1}$). В этом случае $p_{i,j}$ может быть равно $p_{i-1,j-1}$.

Человек с номером i не выбрался из ямы. В этом случае $p_{i,j}$ может быть равно $p_{i-1,j}$.

Итого $p_{i,j}$ равно максимуму из двух вариантов и может быть посчитано за O(1). Таким образом, все значения $p_{i,j}$ можно вычислить за $O(N^2)$.

Ответ на задачу равен максимальному j, для которого $p_{N,j}$ не равно минус бесконечности.

Менее эффективные решения оценивались из меньшего числа баллов.

Например, существуют решения, в котором в качестве параметра динамического программирования используются высота яма или сумма ростов людей. Например, можно было строить колонну из людей снизу вверх. При этом люди обрабатываются по убыванию рослости, и подсчитывается максимально возможная глубина ямы $a_{i,j,k}$, из которой могут выбраться первые j человек из первых i, если суммарный рост выбравшихся людей равен k. При этом также требуется рассмотреть два варианта.

Если i-й человек выбирается, то $a_{i,j,k}$ может быть равно $a_{i-1,j,k}$.

Если i-й человек не выбирается, то его рослость минимальна из рассмотренных и он стоит на самом верху колонны. Таким образом, $a_{i,j,k}$ может быть равно $\min(a_{i-1,j-1,k-hi}, k+l_i)$.

Значения $a_{i,j,k}$ также могут быть вычислены динамическим программированием за $O(N^2P)$. При этом ответом является максимальное j, такое что $(P-k)+a_{N,j,k} > H$. Такое решение набирало около 50 баллов.

Также можно было подсчитывать $b_{i,j,k}$ — минимальный суммарный рост выбравшихся j людей из первых i, которые могут выбраться из ямы глубины k. Эту величину так же можно пересчитывать динамическим программированием. При этом, требуется $O(N^2H)$ времени. Такое решение набирало 70 баллов.

Задача 4. Автобусы

Прежде всего, определим, в каких случаях требуется конечное число автобусов. Построим ориентированный граф, вершинами которого будут города, а ребрами – рейсы автобусов. Чтобы автобусы не скапливались в городе, количество прибывающих и отъезжающих автобусов должно совпадать, поэтому в полученном графе количество входящих ребер должно совпадать с количеством исходящих ребер. Если это свойство не выполняется, значит автобусов бесконечное число.

С этого момента будет решать задачу в предположении, что количество автобусов конечно. Сначала разберем случай, когда отсутствуют рейсы, которые захватывают полночь. Изначально будем считать, что во всех городах общее количество автобусов равно нулю. Проследим за движением автобусов в течение первого дня. В тот момент, когда приходит время отправления автобуса по расписанию, мы отправляем любой из автобусов, который находится в городе. Если на данный момент в городе автобусы отсутствуют, добавляем новый автобус и увеличиваем ответ на единицу. Можно заметить, что количества автобусов утром и вечером совпадают, так как в течение дня из города уезжает столько же автобусов, сколько в него возвращается. Поэтому дополнительные автобусы для обеспечения движения по расписанию в последующие дни не потребуются.

Рассмотрим второй случай. Будем считать, что каждый из рейсов, ровно в полночь делает остановку в виртуальном N+1 городе. Тогда рейс, для которого время отправления превышает время прибытия, распадается на две части HH:MM – 24:00 и 00:00 – HH:MM. В результате такого преобразования задача сводится к предыдущему случаю. Другой подход заключается в том, что можно изначально на каждом рейсе, который захватывает полночь, запустить по автобусу и после этого свести задачу к предыдущему случаю.

Существует несколько способов реализовывать перечисленные выше идеи. При решении задачи можно отдельно не разбирать случай бесконечного количества автобусов и рейсы, которые захватывают полночь. Однако при этом появляется возможность допустить ошибку, которую в дальнейшем будет трудно найти и исправить.

Правильное решение задачи требует $O(N + M)$ памяти и работает за время $O(N + M \log M)$. Заведем специальный массив, в котором будем хранить текущее количество автобусов в городе. Каждый рейс преобразуем в два события: отправление и прибытие автобуса. Отсортируем все события по времени их возникновения. Если два события возникают одновременно, сначала будем рассматривать прибытие автобуса, а только потом отправление. Такая модификация позволить автобусу отправиться сразу же после прибытия. Будем просматривать события одно за другим. Если автобус прибывает в город, увеличиваем количество автобусов в этом городе на единицу, если отправляется – уменьшаем на единицу. Если при отправлении, в городе отсутствуют автобусы, увеличиваем ответ на единицу. Понятно, что такой алгоритм будет работать за линейное время, если не учитывать сортировку данных. Это решение набирает 100 баллов.

Можно непосредственно имитировать передвижение автобусов в течение дня, рассматривая дискретные моменты времени с интервалом в одну минуту. Тогда в каждый момент времени автобус либо находится в движении, либо простаивает. Если в указанный момент времени должен отправиться один из автобусов по расписанию, находим автобус, который простаиваем, либо заводим новый автобус. Такое решение на некоторых тестах не укладывается в ограничение по времени и набирает 70 баллов.

Еще одно решение основано на «склеивании» рейсов и выделении циклов в графе. Если есть два рейса из города A в город B и из города B в город C заменяем их одним рейсом из города A в город C, пересчитываем при этом продолжительность рейса. В общем случае через город B может проходить несколько рейсов. В этом случае чтобы общее количество автобусов было минимальным необходимо рейсы разбить на пары так, чтобы минимизировать суммарное время простоя. Продолжаем склейку до тех пор, пока все рейсы не превратятся в петли (циклы). Следует отметить, что в некоторых тестах правильное решение может в одном городе содержать несколько петель и склейка таких маршрутов приводит к ухудшению результата. Это идею можно довести до правильного решения, но трудоемкость этого перехода по сложности такая же, как правильное решение задачи. Решение, которое при склеивании перебирает все пары рейсов, проходящих через заданный город на некоторых тестах не укладывается в ограничения по времени и набирает 70 баллов.

Решение, которое правильно определяет случай бесконечного количества автобусов и проходит небольшие ручные тесты набирает 20 баллов. Решение, которое не учитывает рейсы, захватывающие полночь набирает 40 баллов.

На последок стоит отметить, что решение задачи достаточно сложно отлаживать. Основная проблема в том, что не всегда возможно с первого взгляда найти правильный ответ к задаче. Это может казаться парадоксальным, но в некоторых случаях требуется автобусов больше, чем количество рейсов:

Входные данные	Выходные данные
1 2 1 10:00 2 18:00 2 17:00 1 11:00	3

В общем случае количество автобусов не превышает удвоенного количества рейсов, так как автобус не может простаивать более суток.

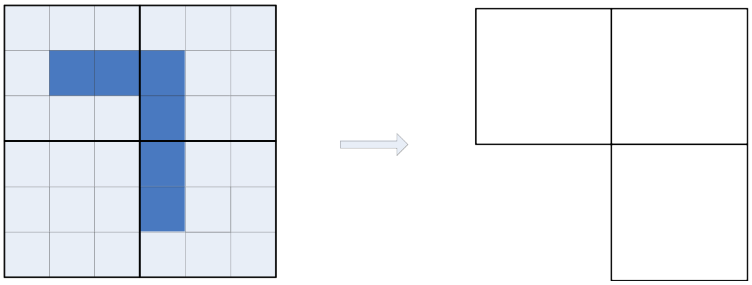
Задача 5. Полимино

Пусть H_1, W_1 —размеры минимального охватывающего прямоугольника нового полимино, N — количество клеток в новом полимино, а H_2, W_2, M — размеры минимального охватывающего прямоугольника старого полимино и количество клеток в нем, соответственно.

Для того, чтобы найти число способов вырезать новое полимино из исходного, увеличенного в K раз, можно перебрать все возможные сдвиги одного относительно другого и проверить что фигуры, приложенные друг к другу таким образом совпадают. Под сдвигом одного полимино относительно другого будем понимать сдвиг фиксированной клетки одного полимино относительно фиксированной клетки другого. Однако, это неэффективное решение с асимптотикой $O(K^2MN)$, которое не укладывается в ограничение по времени.

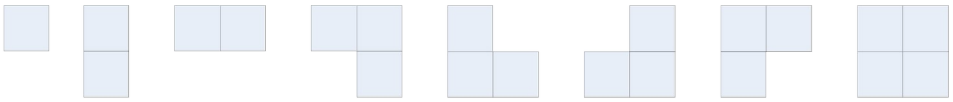
Рассмотрим K^2 возможных сдвигов нового полимино относительно одного из увеличенных квадратов исходного. Для каждого сдвига рассмотрим минимальную конфигурацию клеток, такую, что после увеличения в ней будет встречаться нужное полимино с текущим сдвигом. Назовем такую конфигурацию шаблоном.

Пример: полимино, сдвинутое относительно «клетки» 3 × 3 и шаблон для этого сдвига.

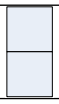


Теперь необходимо лишь проверить, сколько раз встречается этот шаблон в исходном полимино. Количество клеток в шаблоне есть $O(N / K)$, построить его можно за $O(N)$, найти число вхождений шаблона в исходное полимино можно за $O(NM / K)$. Поэтому асимптотика этого решения $O(NMK + K^2N)$

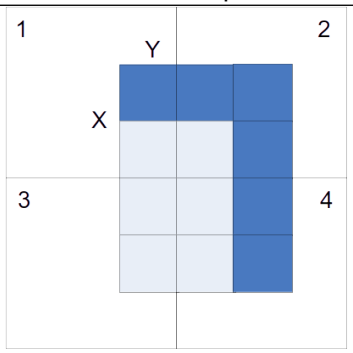
Однако при больших значениях K и это решение не укладывается в ограничение по времени. Нетрудно заметить, что при $K \geq \max(W_1, H_1)$ шаблон для любого сдвига содержит не более 4 клеток. То есть возможны следующие варианты шаблонов:



Для каждого из этих шаблонов за $O(M)$ можно подсчитать количество вхождений его в исходное полимино. Осталось лишь для каждой такой конфигурации найти количество сдвигов, при которых она станет шаблоном для нового полимино. Для первых трех типов шаблонов несложно выписать формулы:

Шаблон:			
Количество сдвигов:	$(K - H_1 + 1)(K - W_1 + 1)$	$(K - W_1 + 1)(H_1 - 1)$	$(K - H_1 + 1)(W_1 - 1)$

Теперь необходимо подсчитать число сдвигов, при которых остальные конфигурации клеток будут шаблонами. Для этого рассмотрим все возможные сдвиги (X, Y) прямоугольника, содержащего полимино, относительно «клетки» размера $K \times K$, так как показано на рисунке ниже.



Число всех возможных сдвигов есть $O(W_1H_1)$. Для каждого такого сдвига можно получить соответствующий шаблон. Для этого необходимо для каждого из четырех прямоугольников определить, есть ли в нем клетки полимино. Это можно сделать за $O(1)$, если предварительно для каждой клетки прямоугольника, содержащего полимино, определить есть ли клетки, находящиеся левее и выше данной, правее и выше, левее и ниже, правее и ниже данной (за $O(W_1H_1)$ для всех клеток прямоугольника). Решение, которое для каждого сдвига строит шаблон за $O(N)$ также укладывается в ограничение по времени.

Пусть A_i – количество сдвигов полимино относительно квадрата $K \times K$, таких, что i-я конфигурация клеток будет шаблоном для каждого из этих сдвигов. Пусть B_i — количество вхождений i-ого шаблона в исходное полимино. Тогда ответом будет сумма $A_i \times B_i$, для i от 1 до 8.

Таким образом, при $K < \max(H_1, W_1)$ асимптотика решения $O(NMK + K^2N)$,

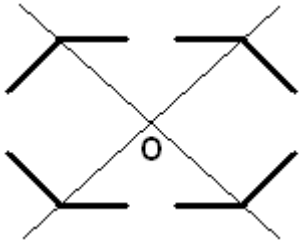
при $K \geq \max(H_1, W_1)$ — $O(W_1H_1 + M)$.

Задача 6. Треугольная реформа

Участникам была предложена задача разрезать многоугольник на минимальное количество треугольников, разрезая только вдоль внутренних диагоналей.

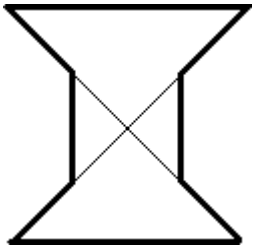
Для начала упомянем, что разрезая вдоль произвольных прямых линий, а не только вдоль диагоналей, невозможно уменьшить количество частей. Ограничение на разрезы было предложено лишь для устранения проблем, связанных с точностью вычислений.

Теперь исследуем, имеет ли смысл разрезать многоугольник вдоль двух пересекающихся внутренних диагоналей:



Предположим, что вдоль одной из диагоналей разрез уже сделан. Тогда мы имеем два отдельных многоугольника, для каждого из которых требуется решить поставленную задачу. Но ни в одном из них точка *O* не является вершиной, а значит, по приведенному выше утверждению, можно решить для них задачу, не проводя разрезы через точку *O*. Значит всегда можно ликвидировать такие пары пересекающихся разрезов.

Впрочем, это не значит, что такие пары разрезов строго увеличивают количество треугольников; вот один из контрпримеров:



Итак, каждая проведенная внутренняя диагональ делит наш многоугольник на два отдельных независимых многоугольника.

Теперь приведем следующие факты из геометрии.

Факт 1. В любом (простом) многоугольнике с более чем тремя вершинами найдется внутренняя диагональ.

Факт 2. Любой *N*-угольник можно разрезать на *N*-2 треугольника.

Построим на базе этих фактов следующий алгоритм.

Алгоритм разделения *N*-угольника на *N*-2 треугольника.

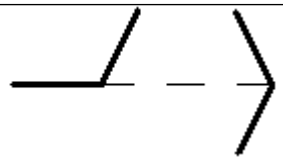
При $N=3$ треугольник сам является единственной искомой областью.

При $N>3$ найдем произвольную внутреннюю диагональ и разрежем многоугольник вдоль нее. Пусть одна из частей оказалась *M*-угольником. Тогда другая часть будет иметь $N-M+2$ вершин (убедитесь в этом!). Запустим наш алгоритм рекурсивно в обеих частях. В одной из них мы получим M-2 треугольника, а в другой — $N-M$ треугольников. Итого имеем N-2 треугольника, что нам и требовалось.

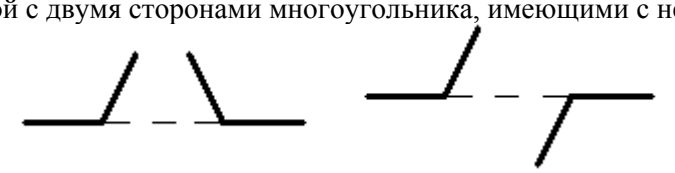
Однако результат работы этого алгоритма может оказаться неоптимальным. Тонкое место в рассуждении – это утверждение о количестве вершин в первой и второй частях. Улучшение может иметь место, если количество вершин в одной или в обеих частей окажется меньше. Это произойдет, если три или четыре вершины, идущие подряд в новом многоугольнике, окажутся на одной прямой. Тогда одну или две из них можно будет перестать рассматривать как вершину.

Это приводит нас к следующим важным определениям.

Определение Диагональю первого типа будем называть внутреннюю диагональ, лежащую на одной прямой ровно с одной стороной многоугольника, имеющей с ней общий конец.



Определение Диагональю второго типа будем называть внутреннюю диагональ, лежащую на одной прямой с двумя сторонами многоугольника, имеющими с ней общий конец.



Определение Диагонали обоих типов будем называть особенными диагоналями.

Таким образом, если в предыдущем алгоритме в какой-то момент выбрать не обычную внутреннюю диагональ, а особенную, то суммарное количество вершин в новых двух многоугольниках уменьшится на номер типа этой диагонали. А значит, на это же число уменьшится и количество треугольников в разбиении.

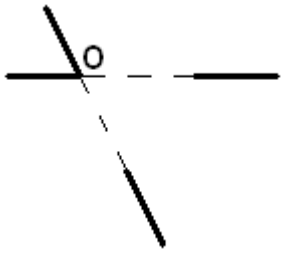
Теперь задача оптимизации видна: в процессе работы алгоритма требуется использовать множество диагоналей с максимальной суммой номеров типов. К сожалению, использовать абсолютно все особенные диагонали получается не всегда: например, две пересекающиеся особенные диагонали использовать вместе невыгодно.

Исследуем подробно все случаи взаимного расположения двух особенных диагоналей.

- Если диагонали пересекаются (строго пересекаются, а не касаются), то их не следует использовать вместе.
- Если диагонали не имеют общих точек, то их можно использовать вместе.

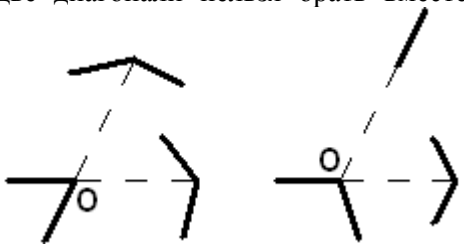
Особенная внимательность требуется в случае, когда две особенные диагонали выходят из одной вершины. В этом случае имеет значение их тип.

- Если это две диагонали второго типа, то их использовать вместе не следует:

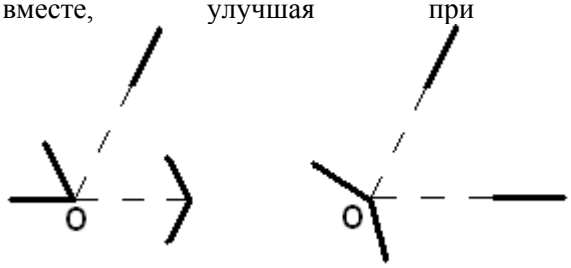


(после разрезания вдоль одной из них точка *O* перестает быть вершиной в новых многоугольниках)

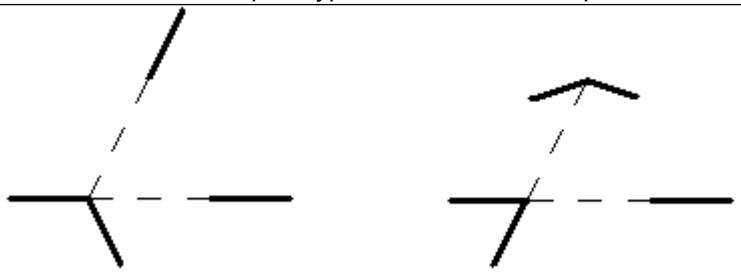
- Для двух диагоналей первого типа есть четыре различных случая. В первых двух случаях две диагонали нельзя брать вместе по той же причине, что и в предыдущем случае:



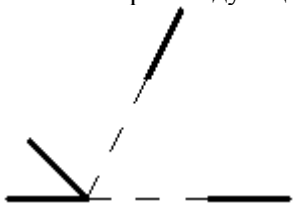
Еще в двух случаях точка *O* является вершиной в обеих частях и диагонали можно брать вместе, улучшая при этом ответ на задачу:



- Если эти две особенные диагонали разных типов, то возникают три различных случая. В первых двух диагонали выбирать вместе нельзя:



И лишь при следующем взаиморасположении диагоналей можно использовать их вместе:



Зная это, поступим следующим образом: построим граф, в котором вершины – это особенные диагонали, а ребра между вершинами проведены тогда, когда их можно использовать вместе. Кроме того, введем вес вершины, равный типу соответствующей особенной диагонали.

Здесь стоит оценить размер этого графа. Каждая особенная диагональ является продолжением некоторой, хотя бы одной, стороны. В то же время, продолжением каждой стороны может являться не более одной особенной диагонали, поскольку иначе появятся две внутренние диагонали многоугольника, лежащие на одной прямой. Отсюда ясно, что количество особенных диагоналей не превышает количества вершин многоугольника.

В построенном графе требуется найти множество вершин, попарно соединенных ребрами между собой, с наибольшим суммарным весом. Эта задача называется “Задачей о максимальной клике” и является NP-полной. Это наталкивает на мысль, что и наша задача не решается за полиномиальное время.

В то же время экспоненциальное решение задачи придумать несложно. Например, можно перебрать все возможные множества вершин в этом графе (их 2^m, где m – количество особенных диагоналей), и для каждого из них проверить, все ли вершины этого множества соединены между собой. Из всех подходящих множеств – выбрать множество с наибольшим весом, оно и будет искомым.

Получив требуемый набор особенных диагоналей, решение изначальной задачи доводится до конца следующим образом:

Разрежем многоугольник вдоль всех особенных диагоналей из найденного набора.

Многоугольник при этом разобьется на некоторое количество меньших многоугольников, в каждом из которых нет особенных диагоналей (иначе их можно было бы добавить в наш максимальный набор). Для каждого из этих многоугольников, следовательно, можно запустить первичный алгоритм, который правильно и оптимально разрежет его на треугольники.

Время работы полученного алгоритма составляет $O(2^N \cdot N^2)$.

Имеется также решение с этой асимптотикой, оформленное в виде динамического программирования.

Рассмотрим некоторый многоугольник, появившийся в процессе разрезания исходного многоугольника внутренними диагоналями. Множество вершин этого многоугольника – подмножество вершин изначального, причем порядок обхода этих вершин определяется однозначно.

Таким образом, имеется не более чем 2^N многоугольников, подлежащих исследованию, и их удобно идентифицировать масками из N битов.

Применим динамическое программирование, чтобы для каждого такого многоугольника подсчитать количество треугольников в его минимальной триангуляции, зная ответы для всех многоугольников с меньшим количеством вершин.

Для треугольника подсчет очевиден.

Для произвольного выпуклого многоугольника подсчет также не представляет сложности.

Теперь рассмотрим некоторый невыпуклый многоугольник. Выберем в нем минимальную по номеру вершину, угол при которой превышает 180°.

В любой триангуляции из этой вершины должна выходить хотя бы одна диагональ. Проведем из нее все возможные внутренние диагонали. В каждом из случаев многоугольник разделится на два других многоугольника, обработанных ранее. Среди всех вариантов выберем вариант с минимальным суммарным количеством треугольников.