Задача A. Перепаковка сумок

Имя входного файла:	`bag-repacking.in`
Имя выходного файла:	`bag-repacking.out`
Ограничение по времени:	2 секунды (*3 секунды для Java*)
Ограничение по памяти:	256 мебибайт

В этой задаче требуется перепаковать набор сумок.

Есть n сумок. Все сумки различны: каждая покрашена в свой уникальный цвет. Каждая сумка довольно вместительна, но сама по себе занимает мало места. На практике это означает, что в любую сумку можно положить все остальные сумки вместе взятые.

Сумки могут лежать одна в другой — либо непосредственно, либо внутри других сумок, также лежащих внутри. Попробуем дать формальное определение этих понятий. Будем говорить, что сумка u *лежит непосредственно* в сумке v, если, открыв сумку v, можно извлечь из неё сумку u, не открывая никакие другие сумки. Будем говорить, что сумка u *находится где-то внутри* сумки v, если либо u лежит непосредственно в v, либо u лежит непосредственно в какой-то сумке w, которая находится где-то внутри сумки v. Каждая сумка может лежать непосредственно не более чем в одной сумке. Если сумка не лежит непосредственно ни в какой другой, будем говорить, что она *находится снаружи*. Никакая сумка не может находиться где-то внутри самой себя.

Будем называть *конфигурацией сумок* информацию для каждой сумки о том, лежит ли она непосредственно в какой-то другой сумке, и если лежит, то в какой именно. Определим две операции, которые можно делать с конфигурацией сумок.

- «`out` u v»: Достать сумку u, которая лежит непосредственно в сумке v, из сумки v, которая находится снаружи.
- «`in` u v»: Положить сумку u, которая находится снаружи, в сумку v, которая также находится снаружи.

Нетрудно видеть, что эти операции — взаимно обратные, а также что любую конфигурацию сумок можно перевести в любую другую некоторой последовательностью таких операций. Конечно, конфигурация сумок должна оставаться корректной после каждой операции, то есть ни одна сумка не должна находиться где-то внутри самой себя.

Переведите исходную конфигурацию сумок в требуемую последовательностью операций, которые описаны выше. Количество операций должно быть минимально возможным.

Формат входных данных

В первой строке ввода задано целое число n — количество сумок ($1 \leqslant n \leqslant 100$). Во второй строке задана исходная конфигурация сумок, а в третьей — требуемая конфигурация. Описание каждой конфигурации сумок состоит из n целых чисел: в какой сумке лежит непосредственно первая, вторая, ..., n-я сумка. Сумки нумеруются целыми числами от 1 до n. Если какая-то сумка находится снаружи, соответствующее число равно нулю. Гарантируется, что обе конфигурации сумок корректны.

Формат выходных данных

В первой строке выведите целое число k — количество операций. Следующие k строк должны описывать сами операции в порядке их произведения. Следуйте формату, указанному в условии. Если возможных ответов несколько, можно вывести любой из них.

Примеры

bag-repacking.in	bag-repacking.out
5 0 1 0 3 1 2 3 0 3 3	5 out 5 1 in 5 3 out 2 1 in 1 2 in 2 3
2 2 0 2 0	0

Пояснения к примерам

В первом примере в исходной конфигурации вторая и пятая сумки лежат непосредственно в первой, а четвёртая — в третьей. Вынем пятую сумку из первой и положим в третью. После этого вынем вторую сумку из первой, положим первую во вторую и, наконец, положим вторую сумку в третью. В итоге первая сумка лежит непосредственно во второй, а вторая, четвёртая и пятая — в третьей. Итак, после пяти указанных операций мы получили требуемую конфигурацию сумок. Можно показать, что меньше чем за пять операций это сделать не получится.

Во втором примере исходная конфигурация совпадает с требуемой: первая сумка находится непосредственно внутри второй. Никаких действий выполнять не требуется.

Задача B. Перестановка битов

Имя входного файла:	`bit-permutation.in`
Имя выходного файла:	`bit-permutation.out`
Ограничение по времени:	2 секунды (*3 секунды для Java*)
Ограничение по памяти:	256 мебибайт

В этой задаче требуется быстро переставлять биты в машинном представлении чисел.

Знаете ли вы, как представляются числа в современных компьютерах? Один из удобных способов хранить не очень большое по модулю целое число — представить его в типе `int32`. В этом типе каждому числу предоставляется в распоряжение 32 бита, нумерующихся целыми числами от 0 до 31. Каждый бит может быть либо нулём, либо единицей. Если значения битов равны b_0, b_1, b_2, …, b_{30}, b_{31}, то само число получается как сумма

$$b_0 \cdot 2^0 + b_1 \cdot 2^1 + b_2 \cdot 2^2 + \ldots + b_{30} \cdot 2^{30} - b_{31} \cdot 2^{31}.$$

Обратите внимание на то, что последнее слагаемое имеет отрицательный знак. В этом типе можно представить любое целое число от -2^{31} до $2^{31} - 1$.

У числа, представленного в типе `int32`, можно переставить биты. Рассмотрим перестановку p_0, p_1, p_2, …, p_{30}, p_{31}, состоящую из целых чисел от 0 до 31, каждое из которых встречается в ней ровно один раз. После перестановки битов x_0, …, x_{31} числа x в соответствии с p получается число $y = p(x)$, состоящее из битов $y_0 = x_{p_0}$, …, $y_{31} = x_{p_{31}}$.

Знаете ли вы, откуда берутся «случайные» числа? Один из простых способов получения псевдослучайных чисел — линейный конгруэнтный генератор. Такой генератор характеризуется константами a (множитель), c (добавка) и m (модуль), а также состоянием s. При генерации следующего числа сначала производится операция $s \leftarrow (s \cdot a + c) \bmod m$, а затем новое значение состояния s объявляется следующим псевдослучайным числом. Конечно, у такого генератора есть период, не превосходящий m: как только в s получилось число, которое уже встречалось ранее, все дальнейшие действия будут давать те же результаты, что и раньше.

Для ускорения работы такого генератора можно избавиться от операции взятия остатка по модулю m. Числа, получающиеся после операции $s \leftarrow (s \cdot a + c)$, будем представлять в типе `int32`. При этом некоторая часть числа может теряться, но у того, что останется, будет такой же остаток от деления на 2^{32}, что и у настоящего результата. Фактически при таком использовании можно считать, что $m = 2^{32}$, но из верхней половины возможных остатков мы вычитаем 2^{32}.

Заданы числа n, a, c и s, а также перестановка битов p. Используя линейный конгруэнтный генератор в типе `int32` с параметрами a и c и начальным состоянием s, сгенерируйте следующие n псевдослучайных чисел x_1, x_2, …, x_n. К каждому из этих чисел примените перестановку битов p, после чего сложите все полученные числа. Будьте внимательны: несмотря на то, что каждое из полученных чисел представимо в типе `int32`, их сумма может не быть в нём представима, поэтому для суммирования следует использовать более широкий тип данных.

Формат входных данных

В первой строке ввода заданы четыре целых числа n, a, c и s — количество операций и параметры линейного конгруэнтного генератора ($1 \leqslant n \leqslant 100\,000\,000$, а числа a, c и s могут быть любыми представимыми в типе `int32`). Гарантируется, что период генератора равен 2^{32}. Во второй строке задана перестановка битов p — числа p_0, p_1, …, p_{31}, среди которых каждое

целое число от 0 до 31 встречается ровно один раз. Соседние числа в строках разделяются пробелом.

Формат выходных данных

Выведите одно целое число: сумму n полученных псевдослучайных чисел, к каждому из которых применена перестановка битов p.

Пример

bit-permutation.in
3 1664525 1013904223 1 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 0
bit-permutation.out
944619915

Пояснение к примеру

В примере получаются следующие случайные числа и перестановки их битов:

$x_1 = 1\,015\,568\,748, \quad p(x_1) = 2\,031\,137\,496;$
$x_2 = 1\,586\,005\,467, \quad p(x_2) = -1\,122\,956\,362;$
$x_3 = -2\,129\,264\,258, \quad p(x_3) = 36\,438\,781.$

Сумма $p(x_1) + p(x_2) + p(x_3)$ равна $944\,619\,915$.

Задача C. Криптовалюта

Имя входного файла:	`coin.in`
Имя выходного файла:	`coin.out`
Ограничение по времени:	2 секунды (*3 секунды для Java*)
Ограничение по памяти:	256 мебибайт

Как известно, особую популярность в последнее время набирают *криптовалюты* — финансовые инструменты, призванные заменить традиционное золото и доллары, и предлагающие децентрализованность, надёжность и отсутствие государственного контроля.

В процессе «добычи» одной из криптовалют используется следующий генератор. Сначала выбирается простое число n. Далее вычисляется число $x = \lfloor \sqrt{n} \rfloor$. После этого вычисляется последовательность $a_i = (x + i)^2 \bmod n$. Каждое k-гладкое число в этой последовательности является очередным ключом, определяющим «добытую» единицу валюты.

Напомним, что число q называется k-гладким, если оно представимо в виде произведения простых чисел с номерами не более k. Например, число $28 = 2^2 \cdot 7 = p_1^2 \cdot p_4$ является 4-гладким и 5-гладким, но не является 3-гладким.

Ваша задача — вывести первые m ключей в данной последовательности.

Формат входных данных

В единственной строке входного файла содержится три целых числа: n, m и k ($10^{18} \leqslant n \leqslant 2 \cdot 10^{18}$, n — простое, $1 \leqslant m \leqslant 5000$, $1000 \leqslant k \leqslant 2000$). Гарантируется, что n либо равно простому числу из примера, либо выбрано случайным образом.

Формат выходных данных

Выведите m чисел: первые m ключей в порядке их появления.

Пример

coin.in
1000000000000000003 5 1500
coin.out
24000000141 76000001441 94000002206 124000003841 160000006397

Задача D. Гирлянды

Имя входного файла:	`garlands.in`
Имя выходного файла:	`garlands.out`
Ограничение по времени:	2 секунды (*3 секунды для Java*)
Ограничение по памяти:	256 мебибайт

Скоро Новый год! Все дома и лавочки в округе украшены свечками и огоньками.

Вы украшаете ёлку гирляндами. В Петербурге довольно ветрено, поэтому вы решили закрепить их несколькими гвоздями, чтобы они не улетели.

Формально, ёлка является деревом из n вершин. Каждая гирлянда — её поддерево. Гвозди можно прибивать в вершинах дерева.

Каждую гирлянду нужно закрепить хотя бы одним гвоздём. В то же время, один гвоздь закрепляет все гирлянды, проходящие через него.

Ваша задача — закрепить все гирлянды минимальным числом гвоздей.

Формат входных данных

В первой строке ввода записано целое число n — количество вершин дерева ($1 \leqslant n \leqslant 100\,000$). Во второй строке записано $n - 1$ целое число $p_2, \ldots, p_n$ ($1 \leqslant p_i < i$). Эти числа описывают рёбра (i, p_i).

В третьей строке записано целое число g — количество гирлянд. Следующие k пар строк описывают сами гирлянды. Каждое описание начинается строкой с целым числом k_i ($1 \leqslant k_i \leqslant n$), количеством вершин в гирлянде. Во второй строке описания записаны k_i различных целых чисел $c_{i,j}$ ($1 \leqslant c_{i,j} \leqslant n$ для $1 \leqslant j \leqslant k_i$) — номера вершин дерева, соответствующих гирлянде. Гарантируется, что эти вершины образуют поддерево: если оставить в дереве только их, полученный подграф будет связным. Также гарантируется, что сумма всех k_i не превосходит 100 000.

Формат выходных данных

В первой строке запишите целое число x — минимальное количество гвоздей.

Во второй строке запишите x различных целых чисел — номера вершин, в которых нужно закрепить гирлянды гвоздями.

Если существует несколько оптимальных решений, выведите любое из них.

Пример

garlands.in	garlands.out
3 1 1 2 1 1 2 1 3	1 1

Задача E. K путей

Имя входного файла:	kpaths.in
Имя выходного файла:	kpaths.out
Ограничение по времени:	2 секунды (*3 секунды для Java*)
Ограничение по памяти:	256 мебибайт

Постройте ориентированный граф, в котором ровно K различных путей из первой вершины во вторую. При этом общее количество вершин графа не должно превышать $2+2\cdot\log_2 K$. В графе не должно быть циклов, петель и кратных рёбер.

Формат входных данных

В первой строке ввода дано целое число K ($1 \leqslant K \leqslant 10^9$).

Формат выходных данных

В первой строке вывода запишите целое число N ($2 \leqslant N \leqslant 2 + 2 \cdot \log_2 K$) — количество вершин в графе.

В последующих N строках выведите списки смежности всех вершин в следующем формате: в $(i+1)$-й строке выведите количество вершин, в которые идёт ребро из вершины с номером i, а затем — номера этих вершин в порядке возрастания. Вершины графа нумеруйте начиная с единицы. Числа внутри каждой строки разделяйте пробелами.

Если существует несколько решений, выведите любое из них.

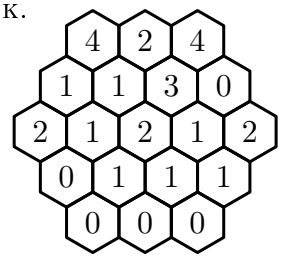
Примеры

kpaths.in	kpaths.out
1	2 1 2 0
4	6 4 3 4 5 6 0 1 2 1 2 1 2 1 2

Задача F. Место встречи

Имя входного файла:	`meeting.in`
Имя выходного файла:	`meeting.out`
Ограничение по времени:	2 секунды (*3 секунды для Java*)
Ограничение по памяти:	256 мебибайт

В этой задаче требуется выбрать место встречи для шестерых человек.



Карта плато Экси — решётка, состоящая из равных правильных шестиугольников. Они образуют один большой «шестиугольник», каждая «сторона» которого состоит из n маленьких шестиугольников. В каждом маленьком шестиугольнике написано число — высота соответствующего участка плато. На самом деле плато Экси состоит из правильных шестиугольных призм, нижние основания которых находятся в одной плоскости, а высоты соответствуют числам на карте. Пример карты для $n = 3$ показан на рисунке справа.

В шести угловых шестиугольниках начинают путь шесть человек. Они хотят выбрать какой-нибудь один шестиугольник на карте и встретиться там.

Человек может переместиться из шестиугольника в любой соседний по стороне, но, когда высоты двух шестиугольников сильно отличаются, сделать это непросто. *Сложностью* перехода в соседний шестиугольник считается модуль разности высот исходного и соседнего шестиугольников.

Путь по плато — это последовательность переходов из шестиугольника в соседний. *Длина пути* равна количеству переходов между шестиугольниками в этом пути. *Сложность пути* равна сумме сложностей всех переходов в нём.

После выбора места встречи каждый человек идёт в него по пути, имеющему наименьшую возможную длину. Изо всех таких путей он выбирает равновероятно один случайный путь. Все шесть человек выбирают пути независимо.

Выберите место встречи так, чтобы минимизировать математическое ожидание суммарной сложности путей, по которым пойдут шесть человек. Иными словами, следует минимизировать сумму шести средних сложностей путей. Средняя сложность пути для одного человека — это сумма сложностей всех путей, которые он может выбрать, делённая на количество этих путей.

Формат входных данных

В первой строке ввода задано целое число n — количество маленьких шестиугольников на «стороне» большого «шестиугольника» ($2 \leqslant n \leqslant 200$). Следующие $(2 \cdot n - 1)$ строк задают карту высот. Каждая высота является целым числом от 0 до 9. Соседние числа отделяются друг от друга пробелом. Будьте внимательны: для наглядного представления карты в начале некоторых строк также находятся пробелы! Для лучшего понимания того, как числа расставляются на карте, ознакомьтесь с поясняющими картинками к примерам.

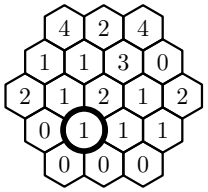
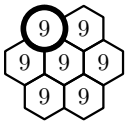
Формат выходных данных

Выведите два целых числа в строке — координаты выбранного места встречи. Первое число — номер строки карты, второе — номер шестиугольника в этой строке. Строки карты нумеруются с единицы сверху вниз. Шестиугольники в каждой строке нумеруются с единицы слева направо. Будьте внимательны: нумерация шестиугольников задаётся отдельно в каждой строке!

Если правильных ответов несколько, можно вывести любой из них. Ответ считается верным, если математическое ожидание суммарной сложности путей отличается от минимально

возможного не более чем на 10^{-6} по абсолютной величине.

Примеры

meeting.in	meeting.out	Пояснение
3 4 2 4 1 1 3 0 2 1 2 1 2 0 1 1 1 0 0 0	4 2	
2 9 9 9 9 9 9 9	1 1	

Пояснения к примерам

В первом примере выбран второй шестиугольник в четвёртой строке. Выпишем сложности всех кратчайших путей в него из каждого углового шестиугольника и найдём среднюю сложность для каждого угла.

- Из левого верхнего угла: $(3 + 3 + 5)/3 = \frac{11}{3}$.
- Из правого верхнего угла: $3/1 = 3$.
- Из правого угла: $(1 + 1 + 3)/3 = \frac{5}{3}$.
- Из правого нижнего угла: $(1 + 1)/2 = 1$.
- Из левого нижнего угла: $1/1 = 1$.
- Из левого угла: $(1 + 3)/2 = 2$.

Сумма этих шести средних сложностей равна $12 + \frac{1}{3}$. Можно убедиться, что, если выбрать для встречи любой другой шестиугольник, математическое ожидание суммарной сложности путей окажется больше.

Во втором примере можно выбрать любой шестиугольник, так как сложность любого пути по плато равна нулю. Обратите внимание: разрешается выбрать шестиугольник, на котором изначально стоит человек.

Задача G. Шаблоны

Имя входного файла:	`patterns.in`
Имя выходного файла:	`patterns.out`
Ограничение по времени:	2 секунды (*3 секунды для Java*)
Ограничение по памяти:	256 мебибайт

Многие приложения используют шаблоны для поиска строк. Приведём одно из простейших определений шаблона.

Шаблоном называется строка из строчных английских букв и звёздочек («*»). Строка s считается *подходящей* под шаблон p тогда и только тогда, когда каждую звёздочку можно заменить на (возможно, пустую) строку таким образом, что получающаяся строка совпадает с s. Различные вхождения звёздочки могут соответствовать как различным, так и совпадающим строкам.

По данным шаблонам p_1 и p_2 найдите строку s, которая подходит под оба из них.

Строка s не должна быть пустой.

Формат входных данных

В первой строке ввода записан шаблон p_1. Во второй строке ввода записан шаблон p_2. Обе строки непусты, а их длины не превышают $2 \cdot 10^5$ каждая. Гарантируется, что обе строки состоят только из строчных английских букв и звёздочек.

Формат выходных данных

В единственной строке вывода запишите либо строку s, подходящую под оба шаблона, либо слово «`Impossible`», если такой строки не существует. Строка s должна состоять только из строчных английских букв. Длина s должна лежать в пределах от 1 до 10^6 символов включительно.

Примеры

patterns.in	patterns.out
a a	a
a b	Impossible
* b	b

Задача H. WTF-8

Имя входного файла:	`wtf8.in`
Имя выходного файла:	`wtf8.out`
Ограничение по времени:	2 секунды (*3 секунды для Java*)
Ограничение по памяти:	256 мебибайт

Дана строка

Нет, вам даны несколько последовательных байт из *WTF-8-кодированной* строки. Аббревиатура "WTF" расшифровывается как «Wonderful Text Format».

Кодировка WTF-8 похожа на UTF-8, но **не совпадает** с ней полностью. Если вы знакомы с UTF-8, не путайтесь. В любом случае, внимательно прочитайте условие.

Каждый символ кодируется следующим образом. Во-первых, он заменяется на свой численный код в соответствии с таблицей. Эта таблица не дана в условии, и она не будет использоваться. Численный код — это целое число без знака $0 \leqslant x < 2^{31}$.

Далее, мы выбираем кратчайшее возможное представление этого кода. Каждое возможное представление — это последовательность байт. В первом байте $0 \leqslant y < 7$ старших бит установлены в 1, после чего следует 0. Число y — это длина представления в байтах.

Если $y = 0$, первый байт является единственным в представлении, и оно соответствует коду $x < 2^7$, равному этому байту. Иначе представление содержит ещё $y - 1$ байт. Старший бит каждого из оставшихся байт установлен в 1, а второй по старшинству — в 0. Код, которому соответствует это представление, равен значению двоичного числа, получаемого конкатенацией $8 - y$ младших бит первого байта и 6 младших бит каждого из оставшихся байт.

Например, код $36 = 100100_2$ представляется одним байтом `00100100`. Код $674 = 1010100010_2$ представляется байтами `11001010 10100010`, код $8364 = 10000010101100_2$ представляется как `11100010 10000010 10101100`.

Ваша задача — вычислить сумму кодов всех символов во входной строке.

Формат входных данных

Ввод состоит из шестнадцатеричных чисел по две цифры в каждом, представляющих последовательность байт. Байты записаны в строках по 16 штук. Байты внутри строки разделяются пробелами.

Во входных данных будет описано не более 65 536 байт.

Шестнадцатеричные цифры записываются символами от "`0`" до "`9`" (десятичные цифры) и от "`A`" до "`F`" (заглавные латинские буквы).

Формат выходных данных

Выведите одно целое число — сумму всех кодов.

Примеры

wtf8.in	wtf8.out
24 C2 A2 E2 82 AC F0 A4 AD A2	158932

Пояснение к примеру

Во вводе четыре символа: `24`, `C2 A2`, `E2 82 AC` и `F0 A4 AD A2`.