Problem A. Сечение куба

Input file:	cubesec.in
Output file:	cubesec.out
Time limit:	2 секунды
Memory limit:	64 megabytes

Саше надоело решать задачи из учебника про сечение кубов, и он хочет заставить компьютер заниматься этой чёрной работой.

Вам требуется написать программу, находящую сечение заданного куба заданной плоскостью, а также площадь этого сечения.

Input

В первой строке заданы координаты одной из вершин куба x_0, y_0, z_0. В последующих трех строках даны направляющие вектора трех ребер куба, выходящих из этой вершины. В пятой строке дана длина ребра куба.

Шестая строка входного файла содержит числа a, b, c, d, задающие плоскость уравнением $ax + by + cz + d = 0$.

Числа во входном файле — вещественные с максимально возможной точностью. Все числа не превосходят 1 000 по модулю. Длина стороны положительна.

Входные данные корректны и проверка их не требуется.

Output

В выходной файл выведите число вершин сечения k. Последующие k троек чисел — координаты вершин сечения в порядке обхода. Выходной файл завершается площадью сечения. Все числа выводятся с шестью точными знаками после запятой.

Никакие три подряд идущие вершины сечения не должны лежать на одной прямой.

Example

cubesec.in	cubesec.out
0 0 0 0 0 1 0 1 0 1 0 0 1 2 0 0 -1	4 0.5 0 0 0.5 0 1 0.5 1 1 0.5 1 0
0 0 0 0 0 1 0 1 0 1 0 0 1 1 0 0 7	0 0
0 0 0 0 0 1 0 1 0 1 0 0 1 1 0 1 -2	2 1 0 1 1 1 1 0

Problem B. Две кривые

Input file:	curves.in
Output file:	curves.out
Time limit:	2 секунды
Memory limit:	64 megabytes

Кривой второго порядка называется множество точек, удовлетворяющих уравнению $ax^2 + bxy + cy^2 + dx + ey + f = 0$ с произвольными вещественными коэффициентами.

Требуется написать программу, рассчитывающую количество областей, на которые делят плоскость две заданные кривые второго порядка.

Input

Входной файл состоит из двух строк, каждая из которых содержит коэффициенты уравнения соответствующей кривой — целые числа, не превосходящие 1 000 по модулю. Хотя бы одно из чисел в описании любой кривой отлично от нуля.

Output

В выходной файл требуется выдать искомое число областей.

Example

curves.in	curves.out
1 0 1 0 0 -1 1 0 1 0 0 -4	3
1 0 1 0 0 0 1 0 1 -2 -2 2	1

Problem C. Кратчайшее расстояние

Input file:	dist.in
Output file:	dist.out
Time limit:	2 секунды
Memory limit:	64 megabytes

В этой задаче вам надо всего лишь найти расстояния от выделенной вершины до всех остальных вершин связного неориентированного графа, в котором не более 26 вершин.

Правда, сам граф задается слегка необычно. А именно, сначала задается дерево с выделенной вершиной, все вершины которого помечены символами от A до Z, а затем вершины с одинаковыми метками отождествляются.

Деревья указанного вида описываются следующим образом:

$\langle letter \rangle ::= \mathbf{A}|\mathbf{B}|...|\mathbf{Z}$
$\langle list \rangle ::= \langle tree \rangle | \langle tree \rangle [, \langle list \rangle]$
$\langle forest \rangle ::= \langle tree \rangle | (\langle list \rangle)$
$\langle tree \rangle ::= \langle letter \rangle | (\langle tree \rangle) | \langle tree \rangle \rightarrow \langle forest \rangle$

Операция $\rightarrow$ выполняется справа налево, то есть $A \rightarrow B \rightarrow C = A \rightarrow (B \rightarrow C)$. При этом любая буква α означает дерево с одной вершиной, помеченной символом α, а $x \rightarrow (y_1, y_2, \ldots, y_s)$ означает дерево, полученное в результате соединения выделенной вершины дерева x с выделенными вершинами каждого из деревьев y_j; выделенной вершиной в результирующем дереве является выделенная вершина дерева x.

Input

На единственной строке входного файла содержится описание графа — строка из символов A...Z, '-', '>', ',', '(' и ')'.

Output

Для каждой из вершин данного графа следует выдать на отдельной строке метку этой вершины и расстояние от нее до выделенной вершины, разделенные пробелом.

Строки следует упорядочить в лексикографическом порядке.

Example

dist.in	dist.out
R->(B->X->S->Y->Z,C->Z)	B 1 C 1 R 0 S 3 X 2 Y 3 Z 2
(A->B)->C	A 0 B 1 C 1

Problem D. Деление уголком

Input file:	division.in
Output file:	division.out
Time limit:	2 секунды
Memory limit:	64 megabytes

Вы отправились на машине времени в далёкое прошлое Франции с важной исторической миссией. Для укрепления своего положения в обществе Вам необходимо получить ученую степень.

Явившись в ближайший университет, Вы с удивлением обнаружили, что для этого необходимо всего лишь продемонстрировать умение делить числа уголком. Для Вас это просто, но для стопроцентной гарантии Вы решили попросить компьютер проделать аналогичные вычисления.

Напишите программу, которая выводит процесс деления двух десятичных чисел уголком.

Input

В двух строках входного файла заданы делимое и делитель, меньшие 10^{100}.

Output

В выходной файл необходимо вывести процесс деления. В точности следуйте формату примера. Никакие числа в процессе вывода не могут иметь ведущие нули. Сравнение будет производиться посимвольно. Делимое должно быть отделено от вертикальной черты ровно одним пробелом.

Example

division.in	division.out
50082003928 491	50082003928 |491 491 +--------- ----- |102000008 982 982 --------- 3928 3928 ---- 0
239 717	239 |717 +--- |0
239 17	239 |17 17 +-- --- |14 69 68 -- 1
667700 6677	667700 |6677 6677 +---- ------ |100 0
12 7	12 |7 7 +- -- |1 5

Problem E. Преобразование Фурье

Input file:	**fourier.in**
Output file:	**fourier.out**
Time limit:	3 секунды
Memory limit:	64 megabytes

Петя недавно нашел в книжке определение *частичного преобразования Фурье* набора комплексных чисел. Теперь он хочет, чтобы Вы написали ему программу для вычисления частичных преобразований Фурье, для того, чтобы получше изучить их свойства.

Вот определение частичного дискретного преобразования Фурье, найденное Петей в книжке:

> Пусть $n = 2^s$, где $s \geq 0$ — целое число. Определим сначала инволюцию rev_s на множестве целых чисел от 0 до $2^s - 1$ следующим образом: $\mathrm{rev}_0(0) = 0$, $\mathrm{rev}_{s+1}(2x) = \mathrm{rev}_s(x)$, $\mathrm{rev}_{s+1}(2x+1) = 2^s + \mathrm{rev}_s(x)$ при $0 \leq x < 2^s$.
>
> Зафиксируем теперь первообразный корень из единицы n-ой степени $\zeta = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$, и определим для любого t от 0 до s *t-ое частичное преобразование Фурье* $\mathrm{Four}_s^{(t)}(\mathbf{a})$ набора $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$ из n комплексных чисел следующим образом: $\mathrm{Four}_s^{(t)}(\mathbf{a}) = \mathbf{b}^{(t)} = (b_0^{(t)}, b_1^{(t)}, \ldots, b_{n-1}^{(t)})$, где
>
> $$b^{(t)}_{2^{s-t}j+k} = \sum_{j'=0}^{2^t-1} \zeta^{2^{s-t}jj'} a_{2^{s-t}j'+\mathrm{rev}_{s-t}(k)}$$
>
> при $0 \leq j < 2^t$, $0 \leq k < 2^{s-t}$
>
> Таким образом, $\mathrm{Four}_s^{(s)} = \mathrm{Four}_s$ — обычное преобразование Фурье:
>
> $$b_j^{(s)} = \sum_{j'=0}^{n-1} \zeta^{jj'} a_{j'} \quad ,$$
>
> а $\mathrm{Four}_s^{(0)}$ всего лишь переставляет числа в исходном наборе: $b_k^{(0)} = a_{\mathrm{rev}_s(k)}$.

У Пети есть подозрение, что найти $\mathrm{Four}_s^{(t)}$ очень просто, если уже известно $\mathrm{Four}_s^{(t-1)}$, однако сам он вывести нужную формулу пока не сумел.

Ваша задача состоит в том, чтобы вычислить t-ое частичное преобразование Фурье данного набора из 2^s комплексных чисел.

Input

Первая строка входного файла содержит числа s и t, разделенные пробелом ($0 \leq t \leq s \leq 16$). Последующие 2^s строк содержат по два вещественных числа каждая — вещественную и мнимую часть соответствующего комплексного числа a_j.

Output

На каждой из 2^s строк выходного файла следует вывести два вещественных числа с восемью цифрами после запятой — вещественную и мнимую часть числа $b_j^{(t)}$.

Example

fourier.in	fourier.out
2 1 2.00 0.00 7.00 0.00 3.00 0.00 -3.00 0.00	5.00000000 0.00000000 4.00000000 0.00000000 -1.00000000 0.00000000 10.00000000 0.00000000

Problem F. KAR

Input file:	**kar.in**
Output file:	**kar.out**
Time limit:	2 секунды
Memory limit:	64 megabytes

Пете пришел по почте файл с расширением KAR. Оказалось, что этот файл запакован новым архиватором Kitten ARchiver. К сожалению, Пете не удалось найти распаковщик для этого архиватора, поэтому он попросил Вас написать такой распаковщик.

Архиватор KAR для паковки файлов использует собственную реализацию популярного метода LZHuf, который заключается в последовательном применении сначала LZSS, а затем кодирования Хаффмана.

Метод LZSS получает на входе файл, состоящий из N символов, а на выходе выдает последовательность из символов и ссылок. Символ в выходе обозначает сам себя, а ссылка имеет вид (x, l) и обозначает копирование l символов, начиная с позиции на x символов левее текущей.

Например, один из возможных способов запаковки строки **abacabadabacaba** выглядит как **ab**$(2,1)$**c**$(4,3)$**d**$(4,2)(8,5)$, а для строки **abababbbab** выход может выглядеть как **ab**$(2,4)(1,2)(8,2)$. В любой ссылке x должен быть строго положительным целым числом, меньшим либо равным количеству запакованных символов во входе, соответствующих выходу до этой ссылки. Очевидно, что это условие гарантирует однозначность распаковки.

Метод Хаффмана также получает последовательность символов на входе. На выходе же он выдает последовательность битов. Символы входного алфавита организуются в двоичное дерево. На каждом из листьев дерева записан какой-то из символов, причем любой символ записан на каком-то листе. На каждом ребре дерева стоит пометка 0 или 1, причём из любой вершины, не являющейся листом, выходит

ровно два ребра, на одном из которых стоит пометка 0, а на другом — 1. Символ кодируется последовательностью битов, соответствующей пути из корня в эту вершину.

Так как выход метода LZSS не является последовательностью символов, требуется некоторое преобразование для использования метода Хаффмана. Оно осуществляется следующим образом. Рассматривается последовательность $a_0 = 1, a_1 = 2, \ldots, a_{2k} = 2^k + 2^{k-1}, a_{2k+1} = 2^{k+1}, \ldots$. Далее фиксируется такое k, что все x и l строго меньше a_k в любом архиве. Для архиватора KAR $k = 32$. После этого для кодирования ссылки используются два класса служебных (не входящих в исходный алфавит) символов X_i и L_j, где $0 \le i, j < k$ — такие числа, что $a_i \le x < a_{i+1}$ и $a_j \le l < a_{j+1}$.

При кодировании методом Хаффмана ссылки на вход сначала подается символ X_i. Затем в обход метода Хаффмана непосредственно в выходной поток выдается *остаток* битовой последовательности — $\log_2(a_{i+1} - a_i)$ битов числа $x - a_i$, от старших к младшим.

После этого на вход методу Хаффмана подается L_j, и аналогичным образом непосредственно в выходной поток выдается остаток. Так как символы L_j всегда идут после символов X_i, то они организуются в отдельное дерево Хаффмана.

Символы исходного алфавита из выходного потока LZSS передаются методу Хаффмана без изменений. Таким образом, метод Хаффмана в архиваторе KAR строит два дерева — одно для символов исходного алфавита (с кодами от 0 до 255) и символов X_i, а второе, отдельное, — для символов L_j.

Рассмотрим на примере, как будет кодироваться строчка **abacabadabacaba**, преобразованная LZSS в **ab**$(2,1)$**c**$(4,3)$**d**$(4,2)(8,5)$. Вход метода Хаффмана будет иметь вид **ab**X_1L_0**c**X_3L_2**d**$X_3L_1X_5L_3$. Предположим для упрощения примера, что никаких других символов, кроме встречающихся во входной строке метода Хаффмана, в алфавите нет. Тогда символы первого алфавита могли бы, например, кодироваться следующим образом: **a** — 000, **b** — 001, **c** — 010, **d** — 011, X_1 — 100, X_5 — 101, X_3 — 11; а символы L_j — так: L_0 — 00, L_1 — 01, L_2 — 10, L_3 — 11.

Таким образом, выходной битовый поток после применения LZHuf будет выглядеть следующим образом:

000 001 100 00 010 11 *0* 10 011 11 *0* 01 101 *00* 11 *1*

(для удобства восприятия последовательности, соответствующие разным символам, отделены друг от друга пробелами, а биты, выдающиеся в выходной поток в обход метода Хаффмана, помечены наклонным шрифтом).

Сам архив KAR имеет следующий формат. Первые четыре байта — сигнатура архива. Она имеет фиксированный вид — это текстовая строка `KAr`, завершенная символом с кодом 4. В Hex-Dump это выглядит как `4B 41 72 04`. Затем записана четырехбайтовая целая длина исходного файла в формате Little-Endian (то есть, байты расположены от младших к старшим, а биты — от старших к младшим, например, число 239 017 будет в этом формате иметь вид `A9 A5 03 00`.

Далее следует таблица кодов Хаффмана, закодированная по Хаффману. В ней для каждого символа задается глубина его положения в дереве Хаффмана. Представление Хаффмана назначается символам по следующему принципу. Вначале находятся символы с максимальной длиной представления Хаффмана l_1. Им назначаются строчки: из l_1 нулей — символу с самым маленьким кодом в исходном алфавите, затем из $l_1 - 1$ нуля и одной единицы — символу со следующим, и т.д. в лексикографическом порядке. Как только заканчиваются символы с длиной представления l_1, выбираются символы со следующей по величине длиной представления l_2. Также берется битовая последовательность для последнего из символов с длиной представления l_1, по ней генерируется лексикографически следующая, и усекается до длины l_2. Для корректности построения дерева Хаффмана требуется, чтобы все отсеченные биты были нулями. Эта усеченная последовательность и является последовательностью символа, код которого минимален из всех, имеющих длину представления l_2. Далее лексикографически следующие за этой последовательностью назначаются следующим символам с длиной представления l_2, и так пока не кончатся символы с такой длиной представления. Далее процесс повторяется. Если записанная длина представления символа равна нулю, то это значит, что символ в выходе LZSS не встречается, и ему не сопоставляется никакое представление.

Например, если есть 6 символов: **x** и **y** с длиной представления 4, X_1, X_2 и X_3 — с длиной представления 3, и X_0 — с длиной представления 1, то коды будут назначаться следующим образом: **x** — 0000, **y** — 0001, X_1 — 001, X_2 — 010, X_3 — 011, X_0 — 1.

Лексикографический порядок символов задается следующим образом: сначала идут символы с кодами от 0 до 255 из исходного алфавита, затем символы от X_0 до X_{31} в порядке возрастания индекса. Символы L_j также лексикографически упорядочены в порядке возрастания индекса.

Гарантируется, что ни в одном из двух алфавитов никогда не будет символов с длиной представления более 23, таким образом, для каждого из символов длину представления можно рассматривать как символ от 0 до 23 (всего 24 символа). Всего таблица длин состоит из 320 таких символов (сначала длины представлений символов первого алфавита, затем — второго). Для этой таблицы тоже строится дерево Хаффмана, и сама таблица кодируется по Хаффману. Для нее записывается вначале таблица длин для каждого из символов от 0 до 23, по 4 бита для каждого символа, слева направо (таким образом, длины представлений длин представлений не превосходят 15). Последующий битовый поток представляет собой закодированные длины сначала для первого дерева (символы с кодами от 0 до 255 и X_i), а затем для второго (символы L_j).

Затем идет непосредственно битовый поток, получившийся после обработки исходного файла методом LZHuf. Биты идут слева направо (от старших битов числа к младшим). Если в самый последний байт ушло нецелое число битов, то справа дописываются нули.

Требуется восстановить по архиву KAR исходный файл. Гарантируется, что это можно сделать однозначным образом. Также гарантируется, что длина исходного файла положительна и не превосходит 239 017 байт.

Input

Hex-Dump архива KAR — байты, записанные двумя шестнадцатеричными цифрами каждый, возможно разделенные пробелами и/или переводами строки.

Output

Строка шестнадцатеричных цифр, представляющая исходный файл. Никаких разделителей между байтами не требуется.

Example

kar.in

```
4B 41 72 04  0F 00 00 00   10 22 00 00  00 00 00 00
00 00 00 00  FF FF FF FF   FF FF FF FF  FF FF FF FF
AA FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF
FF FF FF FF  FB 2F FF FF   FE 01 FF FF  FF E0 C1 69
E6 9C
```

kar.out

```
6162616361626164616261636161626 1
```

kar.in

```
4B 41 72 04  0F 00 00 00   13 44 20 00  00 00 00 00
00 00 00 00  FF DD FF FF   D7 FF 7F FF  FF FB FF FF
FF D7 F0 C7  7D FF FF FF   FF FF FF FF  FF FF FF FF
FF FF FF FF  FF FF FF FF   FF FF 27 FF  FF FF F5 7F
A8 53 63 63  10
```

kar.out

```
48656C6C6F2C20776F726C64210D0A
```

kar.in

```
4B 41 72 04  1A 00 00 00   13 23 00 00  00 00 00 00
00 00 00 00  FF FF FF FF   FF FF 25 FF  FF FF FF FF
FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF
FF FF FF FF  F8 FF FF FF   FF 7A 3F FF  FF E2 1A 3A
```

kar.out

```
3030303131313131313131323232323232323232323232323232
```

Problem G. Художник

Input file:	**painter.in**
Output file:	**painter.out**
Time limit:	4 секунды
Memory limit:	64 megabytes

Итальянский художник-абстракционист Ф. Мандарино увлекся рисованием одномерных черно-белых картин. Он пытается найти оптимальное местоположение и количество черных участков картины. Для этого он проводит на прямой белые и черные отрезки, и после каждой из таких операций хочет знать количество черных отрезков на получившейся картине и их суммарную длину.

Изначально прямая — белая. Ваша задача — написать программу, которая после каждой из таких операций выводит в выходной файл интересующие художника данные.

Input

В первой строке входного файла содержится общее количество нарисованных отрезков ($1 \le N \le 100\,000$). В последующих N строках содержится описание операций. Каждая операция описывается строкой вида $c\ x\ l$, где c — цвет отрезка (`W` для белых отрезков, `B` для черных), а сам отрезок имеет вид $[x; x + l]$, причем координаты обоих концов — целые числа, не превосходящие по модулю 500 000. Длина задается положительным целым числом.

Output

После выполнения каждой из операций необходимо вывести в выходной файл на отдельной строке количество черных отрезков на картине и их суммарную длину, разделенные одним пробелом.

Example

painter.in	painter.out
7 W 2 3 B 2 2 B 4 2 B 3 2 B 7 2 W 3 1 W 0 10	0 0 1 2 1 4 1 4 2 6 3 5 0 0

Problem H. Перемножатель

Input file:	multiply.in
Output file:	multiply.out
Time limit:	2 секунды
Memory limit:	64 megabytes

В хорошо известном казино Лас-Вегаса "Big Jo" недавно появились новые игровые автоматы — "Магические Перемножатели". У них N рычагов и одна большая красная кнопка. Над каждым из рычагов на экране написано целое число a_i в пределах от 1 до M.

Игрок, который хочет сыграть в Перемножатель, сначала вставляет монету в предназначенное для этого гнездо. Затем он выбирает какие-то рычаги и дергает их. После выбора рычагов, игрок нажимает большую красную кнопку. Перемножатель мигает лампочками, звенит и играет различные мелодии, а затем объявляет, победил игрок или нет.

Алгоритм определения выигрыша игрока устроен следующим образом. Если игрок выбрал какое-то подмножество рычагов $S \subset \{1, 2, \ldots, N\}$, его очки — это произведение написанных на них чисел, взятое по модулю M (если игрок не выбрал ни одного рычага, его очки равны 1):

$$score = \prod_{i \in S} a_i \bmod M$$

Игрок победил, если его очки — максимально возможные для данного набора чисел.

По данному описанию Перемножателя определите набор рычагов, который обеспечивает выигрыш.

Input

В первой строке входного файла содержатся два целых числа $1 \le N \le 16$ и $2 \le M \le 2\,000\,000\,000$. Во второй строке записаны N целых чисел $a_1, a_2, \ldots, a_N$ в пределах от 1 до M.

Output

В первой строке выведите одно целое число — сколько очков необходимо набрать для выигрыша. Во второй строке выведите номера рычагов, которые нужно подергать для получения такого числа очков, в возрастающем порядке (нумерация рычагов начинается с единицы).

Если возможны несколько решений, выведите любое.

Example

multiply.in	multiply.out
4 6 1 2 3 4	4 1 4
4 4 4 4 4 4	1

Problem I. Сумма кубов

Input file:	sumcubes.in
Output file:	sumcubes.out
Time limit:	2 секунды
Memory limit:	64 megabytes

Известно, что любое натуральное число можно представить в виде суммы не более чем четырёх квадратов каких-то натуральных чисел. Вася решил придумать аналогичное утверждение для кубов — он хочет узнать, сколько же кубов достаточно для представления любого числа. Первая рабочая гипотеза — восемь.

Выяснилось, что почти все числа, которые Вася смог придумать, представляются в виде суммы не более чем восьми кубов. Однако число 239, например, не допускает такого представления. Вася заинтересовался этим вопросом, и теперь он хочет найти какие-либо другие такие числа, а также, возможно, какую-либо закономерность в представлениях всех остальных чисел, чтобы выдвинуть гипотезу относительно вида всех чисел, которые не представляются в виде суммы восьми кубов.

Помогите Васе написать программу, которая проверяла бы, возможно ли представить данное натуральное число в виде суммы не более чем восьми кубов натуральных чисел, и если это возможно, то находила бы какое-либо такое представление.

Input

Натуральное число $N \le 2\,000\,000\,000$.

Output

Не более восьми натуральных чисел, кубы которых в сумме дают N. Если искомого представления не существует, то в выходной файл необходимо вывести слово `IMPOSSIBLE`.

Example

sumcubes.in	sumcubes.out
17	2 2 1
239	IMPOSSIBLE

Problem J. Сумма двух

Input file:	sumtwo.in
Output file:	sumtwo.out
Time limit:	3 секунды
Memory limit:	64 megabytes

У Пети есть набор из n карточек, на каждой из которых написано какое-то число. Он хочет разложить их в ряд таким образом, чтобы максимум сумм всех пар соседних карточек был минимально возможным.

Input

В первой строке входного файла задано число n ($2 \le n \le 239\,017$). Во второй строке содержится n целых чисел, не превосходящих по модулю 1 000 000 000.

Output

В первой строке выходного файла необходимо вывести минимальное значение максимума сумм двух соседних карточек. Во второй строке выведите одну из возможных конфигураций карточек, доставляющих минимум.

Example

sumtwo.in	sumtwo.out
4 2 3 9 17	19 17 2 3 9