Problem A. Bag Repacking

Input file:	`bag-repacking.in`
Output file:	`bag-repacking.out`
Time limit:	2 seconds (*3 seconds for Java*)
Memory limit:	256 mebibytes

In this problem, you have to repack a collection of bags.

There are n bags to consider. All bags are different: each one is painted in its own unique color. Each bag is capacious but takes little space by itself. In practice it means that for each bag, it is possible to put all other bags in it.

A bag can be contained in another one, either immediately or inside some other bags which are also inside. Let us try to establish formal definitions for these concepts. We say that bag u is *immediately contained* in bag v if, after opening bag v, we can take bag u outside of bag v without opening any other bag. We say that bag u is *located somewhere inside* bag v if either u is immediately contained in v or u is immediately contained in some bag w which is located somewhere inside bag v. Each bag can be immediately contained in at most one bag. If a bag is not immediately contained in any other bag, we say that this bag is *located outside*. No bag can be located somewhere inside itself.

Let a *configuration of bags* be the information for every bag whether it is immediately contained in some other bag, and if it is, in which one. There are two operations on a configuration of bags.

- "`out` u v": Take bag u which is immediately contained in bag v out of bag v which is located outside.
- "`in` u v": Put bag u which is located outside into bag v which is also located outside.

It is easy to see that these two operations are mutually inverse, and also that each possible configuration of bags can be transformed into any other by some sequence of such operations. Naturally, configuration of bags must remain proper after every operation, that is, no bag can be located somewhere inside itself.

Transform the initial configuration of bags into the requested configuration by a sequence of operations defined above. The number of operations must be the minimal possible.

Input

The first line of input contains an integer n, the number of bags ($1 \le n \le 100$). The second line describes the initial configuration of bags, and the third line contains the requested configuration. The description of each configuration consists of n integers: the number of bag in which the first, the second, ..., the n-th bag is contained. The bags are numbered by integers from 1 to n. If a bag is located outside, the respective number is zero. It is guaranteed that both given configurations are proper.

Output

The first line must contain an integer k, the total number of operations. The next k lines must describe the operations themselves in the order of execution. Adhere to the format described in the statement. If there are mupliple possible answers, find any one of them.

Examples

bag-repacking.in	bag-repacking.out

5
0 1 0 3 1
2 3 0 3 3
	5
	out 5 1
	in 5 2
	out 2 1
	in 1 2
	in 2 3

2
2 0
2 0
	0

Explanations

In the first example, in the initial configuration, the second and the fifth bags are immediately contained in the first one, and the fourth bag is immediately contained in the third one. Let us first take bag 5 out of bag 1 and put it into bag 3. After that, take bag 2 out of bag 1, put bag 1 into bag 2, and finally, put bag 2 into bag 3. As a result, the first bag is immediately contained in the second one, and the second, the fourth and the fifth in the third one. So, after five described operations, we got the requested configuration of bags. It can be shown that less than five operations will not suffice to do that.

In the second example, the initial configuration coincides with the requested one: the first bag is immediately contained in the second one. There are no operations to execute.

Problem B. Bit Permutation

Input file:	`bit-permutation.in`
Output file:	`bit-permutation.out`
Time limit:	2 seconds (*3 seconds for Java*)
Memory limit:	256 mebibytes

In this problem, you have to find a fast way to rearrange the bits in a machine integer.

Do you know how numbers are represented in modern hardware? A convenient way to store an integer number which is not too large by the absolute value is to represent it in the `int32` data type. This data type grants 32 bits to store each integer. The bits are numbered from 0 to 31. Each bit can be either zero or one. If the values of the bits are b_0, b_1, b_2, …, b_{30}, b_{31}, the represented number is the sum

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

Please note that the last summand has negative sign. This type can represent any integer number from -2^{31} to $2^{31} - 1$.

One can rearrange the bits of a number represented in `int32` data type. Consider a permutation p_0, p_1, p_2, …, p_{30}, p_{31} consisting of integers from 0 to 31 each of which occurs in the permutation exactly once. After rearranging the bits $x_0, \ldots, x_{31}$ of a number x according to p, one gets $y = p(x)$ consisting of bits $y_0 = x_{p_0}, \ldots, y_{31} = x_{p_{31}}$.

Do you know where "random" numbers come from? A simple way to get pseudorandom numbers is to use a linear congruential generator. Such a generator is characterized by constants a (multiplier), c (increment) and m (modulus) along with the state s. To generate the next pseudorandom number, we first perform the operation $s \leftarrow (s \cdot a + c) \bmod m$. After that, the new value of state s is declared to be the next pseudorandom number. Surely, such generator has a period no greater than m: as soon as s becomes a number which already occurred earlier, all subsequent calculations will have the same results as before.

To speed up such a generator, one can get rid of the modulo m operation. Let use represent the number obtained by $s \leftarrow (s \cdot a + c)$ in `int32` data type. Some part of the number may be lost in the process, but the remaining part has the same remainder modulo 2^{32} as the true result. Actually, the result is the same as using modulus $m = 2^{32}$ and subtracting 2^{32} from the top half of the possible remainders.

You are given numbers n, a, c and s, as well as the bit permutation p. Using a linear congruential generator utilizing `int32` data type with parameters a and c and initial state s, generate the next n pseudorandom numbers x_1, x_2, …, x_n. Apply the permutation p to the bits of each of these numbers and calculate the sum of the resulting values. Please note that, although each of the resulting values can be represented in `int32` data type, their sum does not necessarily have that property, so it is better to use a wider data type for the sum.

Input

The first line of input contains four integers n, a, c and s: the number of operations and the parameters of the linear congruential generator ($1 \le n \le 100\,000\,000$, and the numbers a, c and s can be arbitrary integers which can be represented in `int32` data type). It is guaranteed that the period of the generator is 2^{32}. The secon line contains p, the permutation of bits: numbers p_0, p_1, …, p_{31} among which each integer from 0 to 31 occurs exactly once. Consecutive integers on a line are separated by a space.

Output

Print one integer: the sum of n values each of which is the rearrangement of bits in the next pseudorandom number produced by the generator.

Example

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

Explanation

In the first example, we get the following random numbers and their bit permutations:

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

The sum $p(x_1) + p(x_2) + p(x_3)$ is equal to $944\,619\,915$.

Problem C. Cryptocurrency

Input file:	`coin.in`
Output file:	`coin.out`
Time limit:	2 seconds (*3 seconds for Java*)
Memory limit:	256 mebibytes

As is well known, special popularity was recently gained by *cryptocurrencies*: the financial instruments designed to supersede usual gold and dollars, offering decentralization, safety and no governmental control.

While "mining" one certain cryptocurrency, the following generator is used. First, a prime number n is chosen. Then the number $x = \lfloor \sqrt{n} \rfloor$ is calculated. After that, the sequence $a_i = (x+i)^2 \bmod n$ is obtained. Each k-smooth number in this sequence is the next key that defines the newly "mined" unit of currency.

Recall that a number q is called *k-smooth* if and only if it can be represented as a product of primes with numbers not exceeding k. For example, the number $28 = 2^2 \cdot 7 = p_1^2 \cdot p_4$ is 4-smooth and 5-smooth, but it is not 3-smooth.

Your goal is to output the first m keys in given sequence.

Input

The only line of input contains three integers: n, m and k ($10^{18} \le n \le 2 \cdot 10^{18}$, n is prime, $1 \le m \le 5000$, $1000 \le k \le 2000$). It is guaranteed that the number n is either the same as in the example or chosen randomly.

Output

Output m integers: the first m keys in order of obtaining them.

Example

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

Problem D. Garlands

Input file:	`garlands.in`
Output file:	`garlands.out`
Time limit:	2 seconds (*3 seconds for Java*)
Memory limit:	256 mebibytes

It's Christmas time! Every house and every store around tries to drag your attention with lights and candles.

You are decorating your Christmas tree with several garlands. It's windy in St. Petersburg, so you decide to hammer extra nails in some places to be sure they won't fly away.

To be slightly formal, the Christmas tree is a tree with n vertices. Every garland is a subtree of it. You can place nails in vertices of the tree.

Each garland must be fixed with at least one nail. However, a single nail fixes all garlands at that point.

Your task is to fix all garlands using the minimal possible number of nails.

Input

The first line of the input file contains integer n: the number of vertices ($1 \le n \le 100\,000$). The second line contains $n-1$ integers $p_2, \ldots, p_n$ ($1 \le p_i < i$). Theese numbers describe edges (i, p_i).

The third line contains integer g: the number of garlands. The following k pairs of line describe them. Each garland description starts with an integer k_i ($1 \le k_i \le n$), the number of vertices in the garland. The second line of such description contains k_i distinct integers $c_{i,j}$ ($1 \le c_{i,j} \le n$ for $1 \le j \le k_i$): the vertices covered by the garland. These vertices are guaranteed to form a subtree, that is, if we remove all other vertices from the tree, the remaining part will be connected. It is also guaranteed that the sum of all k_i does not exceed $100\,000$.

Output

On the first line of output, print a single integer x: the minimal number of nails.

On the second line, print x distinct integers: the numbers of vertices to hammer the nails in.

In case of multiple soultions, print any optimal one.

Example

garlands.in	garlands.out

3
1 1
2
1
1
2
1 3
	1
	1

Problem E. K Paths

Input file:	`kpaths.in`
Output file:	`kpaths.out`
Time limit:	2 seconds (*3 seconds for Java*)
Memory limit:	256 mebibytes

Construct an oriented graph such that there are exactly K different paths from the first vertex to the second vertex. The total number of vertices must not exceed $2 + 2 \cdot \log_2 K$. Graph must not contain cycles, loops and multiple edges.

Input

The first line of input contains an integer K ($1 \leq K \leq 10^9$).

Output

On the first line, print an integer N ($2 \leq N \leq 2+2\cdot\log_2 K$), the number of vertices in your graph.

On the next N lines, print the lists of direct successors of all vertices in the following form: on $(i + 1)$-th line, print the number of direct successors of i-th vertex and then the numbers of those successors in ascending order. Vertices are numbered from 1. Separate numbers inside each line by spaces.

If there is more than one solution, print any one of them.

Examples

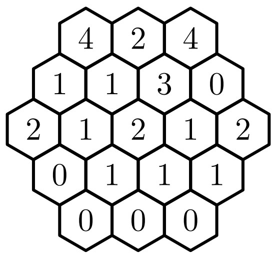
kpaths.in	kpaths.out

1	2
	1 2
	0

4	6
	4 3 4 5 6
	0
	1 2
	1 2
	1 2
	1 2

Problem F. Meeting Point

Input file:	`meeting.in`
Output file:	`meeting.out`
Time limit:	2 seconds (*3 seconds for Java*)
Memory limit:	256 mebibytes



In this problem, you have to pick a meeting point for six people.

The map of Eksi Plateau is a grid consisting of equal regular hexagons. They form a large "hexagon". Each of its "sides" consists of n small hexagons. Each small hexagon contains a number: the height of the respective part of the plateau. In fact, Eksi Plateau consists of regular hexagonal prisms. The lower bases of these prisms lie in one plane, and their heights correspond to the numbers on the map. An example of a map for $n = 3$ is shown on the picture to the right.

Six men start their journeys in six corner hexagons of the plateau. They want to choose a single hexagon on the map and meet there. A man can move from a hexagon to any adjacent one sharing a side with it. However, when the heights of the two hexagons are too different, this is not an easy task. The *complexity* of transition to an adjacent hexagon is the absolute value of the difference of heights of the initial hexagon and the adjacent one.

A *path* on the plateau is a sequence of transitions between adjacent hexagons. The *length of a path* is the number of transitions between hexagons in that path. The *complexity of a path* is the sum of complexities of all transitions in that path.

After choosing the meeting point, each man moves there along a path which has the minimal possible length. From all such paths, he chooses one uniformly at random. All six men choose their paths independently.

Choose the meeting point in such a way that the expected sum of complexities of the six men's paths is minimal possible. That is, you must minimize the sum of six average complexities of paths. An average complexity of a path for one of the men is the sum of complexities of all paths he can choose divided by the number of such paths.

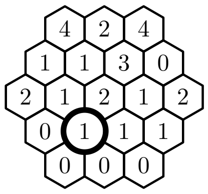
Input

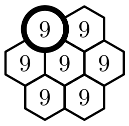
The first line of input contains an integer n, the number of small hexagons on a "side" of the large "hexagon" ($2 \leq n \leq 200$). The next $(2 \cdot n - 1)$ lines define the elevation map. Each elevation is an integer between 0 and 9. Each two consecutive numbers are separated by a single space. Attention: some of these lines also contain spaces at the beginning to make the input more intuitive! For better understanding of how the input numbers are placed on the map, see the explanatory pictures for the examples.

Output

Print two integers on a line: the coordinates of the chosen meeting point. The first integer is the number of the line of the map, the second one is the number of the hexagon in that line. The lines of the map are numbered from one from top to bottom. The hexagons on each line are numbered from one from left to right. Attention: the numbering of hexagons is defined separately for each line!

`meeting.in`	`meeting.out`	Notes

`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`	

Explanations

In the first example, we choose the second hexagon in the fourth line. Let us write down the complexity of each shortest path to that hexagon from each of the corner hexagons, and then find the average complexity for each corner.

- From the upper left corner: $(3 + 3 + 5)/3 = \frac{11}{3}$.
- From the upper right corner: $3/1 = 3$.
- From the rightmost corner: $(1 + 1 + 3)/3 = \frac{5}{3}$.
- From the lower right corner: $(1 + 1)/2 = 1$.
- From the lower left corner: $1/1 = 1$.
- From the leftmost corner: $(1 + 3)/2 = 2$.

The sum of these six average complexities is $12 + \frac{1}{3}$. One can check that, if we choose any other hexagon for the meeting, the expected total complexity of the paths will be greater.

In the second example, we can choose any hexagon since the complexity of each path on the plateau is zero. Please note that we can choose a hexagon which has a man on it initially.

Problem G. Patterns

Input file:	`patterns.in`
Output file:	`patterns.out`
Time limit:	2 seconds (*3 seconds for Java*)
Memory limit:	256 mebibytes

Many applications use patterns to find matching strings. One of the simplest pattern definitions follows.

A *pattern* is a string consisting of lowercase English letters and asterisks (“`*`”). String s is said to *match* pattern p if and only if each asterisk can be replaced with a (possibly empty) string in a way that the resulting string equals s. Distinct occurrences of the asterisk can be replaced with distinct or equal strings.

Given two patterns p_1 and p_2, find a string s that matches both of them.

The string `s` must be non-empty.

Input

The first line of input contains the pattern p_1. The second line of input contains the pattern p_2. Both lines are non-empty, and their lengths do not exceed $2 \cdot 10^5$ each. It is guaranteed that both lines contain only lowercase English letters and asterisks (“`*`”).

Output

In the only line of output, print either a string s matching both patterns or the word “`Impossible`” if such a string does not exist. The string s must consist only of lowercase English letters. The length of s must be between 1 and 10^6 characters, inclusive.

Examples

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

Problem H. WTF-8

Input file:	`wtf8.in`
Output file:	`wtf8.out`
Time limit:	2 seconds (*3 seconds for Java*)
Memory limit:	256 mebibytes

You are given a string

Nope, you are given several consequent bytes of a *WTF-8-encoded* string. Here, "WTF" stands for "Wonderful Text Format".

WTF-8 encoding is similar but **not identical** to UTF-8. If you are familiar with UTF-8, please do not get confused. Anyway, read the statement carefully.

Every character is encoded in the following way. First, it is replaced with its digital code according to a table. This table is not available in the statement, and it will not be used. The digital code is an unsigned integer $0 \le x < 2^{31}$.

Second, we choose the shortest possible representation of the code. Each possible representation is a sequence of bytes. The first byte has $0 \le y < 7$ highest bits set to 1, followed by a bit set to 0. The value y is the number of bytes in the representation.

If $y = 0$, then the first byte is the only byte in the representation, and it represents the code of $x < 2^7$, equal to that byte. Otherwise, there are $y - 1$ more bytes in the representation. Each of the remaining bytes has the highest bit set to 1 and the second highest bit set to 0. The code represented equals to the value of the binary number of concatenated $8 - y$ lowest bits of the first byte and 6 lowest bits of each of the remaining bytes.

For example, the code $36 = 100100_2$ is represented with a single byte `00100100`. The code $674 = 1010100010_2$ is represented with bytes `11001010 10100010`, the code $8364 = 10000010101100_2$ is represented with `11100010 10000010 10101100`.

Your task is to calculate the sum of codes of all characters in the input string.

Input

Input contains one or more hexadecimal numbers with exactly two digits each: the bytes. There are 16 bytes given on each line of input (except maybe the last one). The bytes on a single line are separated by spaces.

There are no more than 65 536 bytes described in the input.

Hexadecimal digits use characters "`0`" through "`9`" (decimal digits) and "`A`" through "`F`" (capital English letters).

Output

Write the only integer number: the sum of all codes.

Examples

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

Explanation

The input bytes contain four characters: `24`, `C2 A2`, `E2 82 AC` and `F0 A4 AD A2`.