Codeforces Round #116 (Div. 2, ACM-ICPC Rules)

A. Defragmentation

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

In this problem you have to implement an algorithm to defragment your hard disk. The hard disk consists of a sequence of clusters, numbered by integers from 1 to n. The disk has m recorded files, the i-th file occupies clusters with numbers $a_{i,1}$, $a_{i,2}$, ..., a_{i,n_i}. These clusters are not necessarily located consecutively on the disk, but the order in which they are given corresponds to their sequence in the file (cluster $a_{i,1}$ contains the first fragment of the i-th file, cluster $a_{i,2}$ has the second fragment, etc.). Also the disc must have one or several clusters which are free from files.

You are permitted to perform operations of copying the contents of cluster number i to cluster number j (i and j must be different). Moreover, if the cluster number j used to keep some information, it is lost forever. Clusters are not cleaned, but after the defragmentation is complete, some of them are simply declared unusable (although they may possibly still contain some fragments of files).

Your task is to use a sequence of copy operations to ensure that each file occupies a contiguous area of memory. Each file should occupy a consecutive cluster section, the files must follow one after another from the beginning of the hard disk. After defragmentation all free (unused) clusters should be at the end of the hard disk. After defragmenting files can be placed in an arbitrary order. Clusters of each file should go consecutively from first to last. See explanatory examples in the notes.

Print the sequence of operations leading to the disk defragmentation. Note that **you do not have to minimize** the number of operations, but it should not exceed $2n$.

Input

The first line contains two integers n and m $(1 \le n, m \le 200)$ — the number of clusters and the number of files, correspondingly. Next m lines contain descriptions of the files. The first number in the line is n_i $(n_i \ge 1)$, the number of clusters occupied by the i-th file. Then follow n_i numbers $a_{i,1}$, $a_{i,2}$, ..., a_{i,n_i} $(1 \le a_{i,j} \le n)$. It is guaranteed that each cluster number occurs not more than once and $\sum_{i=1}^{m} n_i < n$, that is, there exists at least one unused cluster. Numbers on each line are separated by spaces.

Output

In the first line print a single integer k $(0 \le k \le 2n)$ — the number of operations needed to defragment the disk. Next k lines should contain the operations' descriptions as "i j" (copy the contents of the cluster number i to the cluster number j).

Sample test(s)

input
```
7 2
2 1 2
3 3 4 5
```
output
```
0
```

input
```
7 2
2 1 3
3 2 4 5
```
output
```
3
2 6
3 2
6 3
```

Note

Let's say that a disk consists of 8 clusters and contains two files. The first file occupies two clusters and the second file occupies three clusters. Let's look at examples of correct and incorrect positions of files after defragmentation.

№	File 1	File 2	Correct
1	$1, 2$	$3, 4, 5$	Yes
2	$1, 3$	$2, 4, 5$	No
3	$4, 5$	$1, 2, 3$	Yes
4	$2, 1$	$3, 4, 5$	No
5	$1, 2$	$4, 5, 6$	No

Example 2: each file must occupy a contiguous area of memory.

Example 3: the order of files to each other is not important, at first the second file can be written, and then — the first one.

Example 4: violating the order of file fragments to each other is not allowed.

Example 5: unused clusters should be located at the end, and in this example the unused clusters are 3, 7, 8.

B. Divisibility Rules

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

Vasya studies divisibility rules at school. Here are some of them:

- *Divisibility by 2.* A number is divisible by 2 if and only if its last digit is divisible by 2 or in other words, is even.
- *Divisibility by 3.* A number is divisible by 3 if and only if the sum of its digits is divisible by 3.
- *Divisibility by 4.* A number is divisible by 4 if and only if its last two digits form a number that is divisible by 4.
- *Divisibility by 5.* A number is divisible by 5 if and only if its last digit equals 5 or 0.
- *Divisibility by 6.* A number is divisible by 6 if and only if it is divisible by 2 and 3 simultaneously (that is, if the last digit is even and the sum of all digits is divisible by 3).
- *Divisibility by 7.* Vasya doesn't know such divisibility rule.
- *Divisibility by 8.* A number is divisible by 8 if and only if its last three digits form a number that is divisible by 8.
- *Divisibility by 9.* A number is divisible by 9 if and only if the sum of its digits is divisible by 9.
- *Divisibility by 10.* A number is divisible by 10 if and only if its last digit is a zero.
- *Divisibility by 11.* A number is divisible by 11 if and only if the sum of digits on its odd positions either equals to the sum of digits on the even positions, or they differ in a number that is divisible by 11.

Vasya got interested by the fact that some divisibility rules resemble each other. In fact, to check a number's divisibility by 2, 4, 5, 8 and 10 it is enough to check fulfiling some condition for one or several last digits. Vasya calls such rules the **2-type** rules.

If checking divisibility means finding a sum of digits and checking whether the sum is divisible by the given number, then Vasya calls this rule the **3-type** rule (because it works for numbers 3 and 9).

If we need to find the difference between the sum of digits on odd and even positions and check whether the difference is divisible by the given divisor, this rule is called the **11-type** rule (it works for number 11).

In some cases we should divide the divisor into several factors and check whether rules of different types (2-type, 3-type or 11-type) work there. For example, for number 6 we check 2-type and 3-type rules, for number 66 we check all three types. Such mixed divisibility rules are called **6-type** rules.

And finally, there are some numbers for which no rule works: neither 2-type, nor 3-type, nor 11-type, nor 6-type. The least such number is number 7, so we'll say that in such cases the mysterious **7-type** rule works, the one that Vasya hasn't discovered yet.

Vasya's dream is finding divisibility rules for all possible numbers. He isn't going to stop on the decimal numbers only. As there are quite many numbers, ha can't do it all by himself. Vasya asked you to write a program that determines the divisibility rule type in the b-based notation for the given divisor d.

Input

The first input line contains two integers b and d ($2 \le b, d \le 100$) — the notation system base and the divisor. Both numbers are given in the decimal notation.

Output

On the first output line print the type of the rule in the b-based notation system, where the divisor is d: "`2-type`", "`3-type`", "`11-type`", "`6-type`" or "`7-type`". If there are several such types, print the one that goes earlier in the given sequence. If a number belongs to the 2-type, print on the second line the least number of the last b-based digits that we will need to use to check the divisibility.

Sample test(s)

input
```
10 10
```
output
```
2-type
1
```

input
```
2 3
```
output
```
11-type
```

Note

The divisibility rule for number 3 in binary notation looks as follows: "A number is divisible by 3 if and only if the sum of its digits that occupy the even places differs from the sum of digits that occupy the odd places, in a number that is divisible by 3". That's an 11-type rule. For example, $21_{10} = 10101_2$. For it the sum of digits on odd positions equals $1 + 1 + 1 = 3$, an on even positions — $0 + 0 = 0$. The rule works and the number is divisible by 3.

In some notations a number can fit into the 3-type rule and the 11-type rule. In this case the correct answer is "`3-type`".

C. Letter

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

Patrick has just finished writing a message to his sweetheart Stacey when he noticed that the message didn't look fancy. Patrick was nervous while writing the message, so some of the letters there were lowercase and some of them were uppercase.

Patrick believes that a message is *fancy* if any uppercase letter stands to the left of any lowercase one. In other words, this rule describes the strings where first go zero or more uppercase letters, and then — zero or more lowercase letters.

To make the message fancy, Patrick can erase some letter and add the same letter in the same place in the opposite case (that is, he can replace an uppercase letter with the lowercase one and vice versa). Patrick got interested in the following question: what minimum number of actions do we need to make a message fancy? Changing a letter's case in the message counts as one action. Patrick cannot perform any other actions.

Input

The only line of the input contains a non-empty string consisting of uppercase and lowercase letters. The string's length does not exceed 10^5.

Output

Print a single number — the least number of actions needed to make the message *fancy*.

Sample test(s)

input
```
PRuvetSTAaYA
```
output
```
5
```

input
```
OYPROSTIYAOPECHATALSYAPRIVETSTASYA
```
output
```
0
```

input
```
helloworld
```
output
```
0
```

D. Name

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

Everything got unclear to us in a far away constellation Tau Ceti. Specifically, the Taucetians choose names to their children in a very peculiar manner.

Two young parents `abac` and `bbad` think what name to give to their first-born child. They decided that the name will be the permutation of letters of string s. To keep up with the neighbours, they decided to call the baby so that the name was lexicographically strictly larger than the neighbour's son's name t.

On the other hand, they suspect that a name tax will be introduced shortly. According to it, the Taucetians with lexicographically *larger* names will pay *larger* taxes. That's the reason `abac` and `bbad` want to call the newborn so that the name was lexicographically strictly larger than name t and lexicographically minimum at that.

The lexicographical order of strings is the order we are all used to, the "dictionary" order. Such comparison is used in all modern programming languages to compare strings. Formally, a string p of length n is lexicographically less than string q of length m, if one of the two statements is correct:

- $n < m$, and p is the beginning (prefix) of string q (for example, "`aba`" is less than string "`abaa`"),
- $p_1 = q_1, p_2 = q_2, ..., p_{k-1} = q_{k-1}, p_k < q_k$ for some k $(1 \le k \le min(n, m))$, here characters in strings are numbered starting from 1.

Write a program that, given string s and the heighbours' child's name t determines the string that is the result of permutation of letters in s. The string should be lexicographically strictly more than t and also, lexicographically minimum.

Input

The first line contains a non-empty string s $(1 \le |s| \le 5000)$, where $|s|$ is its length. The second line contains a non-empty string t $(1 \le |t| \le 5000)$, where $|t|$ is its length. Both strings consist of lowercase Latin letters.

Output

Print the sought name or `-1` if it doesn't exist.

Sample test(s)

input
```
aad
aac
```
output
```
aad
```

input
```
abad
bob
```
output
```
daab
```

input
```
abc
defg
```
output
```
-1
```

input
```
czaaab
abcdef
```
output
```
abczaa
```

Note

In the first sample the given string s is the sought one, consequently, we do not need to change the letter order there.

E. Cubes

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

Let's imagine that you're playing the following simple computer game. The screen displays n lined-up cubes. Each cube is painted one of m colors. You are allowed to delete not more than k cubes (that do not necessarily go one after another). After that, the remaining cubes join together (so that the gaps are closed) and the system counts the score. The number of points you score equals to the length of the maximum sequence of cubes of the same color that follow consecutively. Write a program that determines the maximum possible number of points you can score.

Remember, you may delete no more than k any cubes. It is allowed not to delete cubes at all.

Input

The first line contains three integers n, m and k ($1 \le n \le 2 \cdot 10^5$, $1 \le m \le 10^5$, $0 \le k < n$). The second line contains n integers from 1 to m — the numbers of cube colors. The numbers of colors are separated by single spaces.

Output

Print the maximum possible number of points you can score.

Sample test(s)

input
```
10 3 2
1 2 1 1 3 2 1 1 2 2
```
output
```
4
```

input
```
10 2 2
1 2 1 2 1 1 2 1 1 2
```
output
```
5
```

input
```
3 1 2
1 1 1
```
output
```
3
```

Note

In the first sample you should delete the fifth and the sixth cubes.

In the second sample you should delete the fourth and the seventh cubes.

In the third sample you shouldn't delete any cubes.

F. Mathematical Analysis Rocks!

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

Students of group 199 have written their lectures dismally. Now an exam on Mathematical Analysis is approaching and something has to be done asap (that is, quickly). Let's number the students of the group from 1 to n. Each student i ($1 \le i \le n$) has a best friend $p[i]$ ($1 \le p[i] \le n$). In fact, each student is a best friend of *exactly one* student. In other words, all $p[i]$ are different. It is possible that the group also has some really "special individuals" for who $i = p[i]$.

Each student wrote exactly one notebook of lecture notes. We know that the students agreed to act by the following algorithm:

- on the first day of revising each student studies his own Mathematical Analysis notes,
- in the morning of each following day each student gives the notebook to his best friend and takes a notebook from the student who calls him the best friend.

Thus, on the second day the student $p[i]$ ($1 \le i \le n$) studies the i-th student's notes, on the third day the notes go to student $p[p[i]]$ and so on. Due to some characteristics of the boys' friendship (see paragraph 1), each day each student has exactly one notebook to study.

You are given two sequences that describe the situation on the third and fourth days of revising:

- $a_1, a_2, ..., a_n$, where a_i means the student who gets the i-th student's notebook on the third day of revising;
- $b_1, b_2, ..., b_n$, where b_i means the student who gets the i-th student's notebook on the fourth day of revising.

You do not know array p, that is you do not know who is the best friend to who. Write a program that finds p by the given sequences a and b.

Input

The first line contains integer n ($1 \le n \le 10^5$) — the number of students in the group. The second line contains sequence of different integers $a_1, a_2, ..., a_n$ ($1 \le a_i \le n$). The third line contains the sequence of different integers $b_1, b_2, ..., b_n$ ($1 \le b_i \le n$).

Output

Print sequence n of different integers $p[1], p[2], ..., p[n]$ ($1 \le p[i] \le n$). It is guaranteed that the solution exists and that it is unique.

Sample test(s)

input

```
4
2 1 4 3
3 4 2 1
```

output

```
4 3 1 2
```

input

```
5
5 2 3 1 4
1 3 2 4 5
```

output

```
4 3 2 5 1
```

input

```
2
1 2
2 1
```

output

```
2 1
```