Contest Ciphers

Cryptography is an exciting discipline, don't you think? Since ancient times, people always felt the need to protect the content of their sensitive messages. Thank to cryptography, a message content (so-called *plaintext*) can be encrypted (*enciphered*) into its encoded form (*ciphertext*) that may be then transferred across insecure channels because it is unreadable for anyone except the intended recipient, who will decrypt (*decipher*) the message into the plaintext again. Good algorithms are parametrized by *keys* — even if we know the algorithm, it is still impossible to read the message without the proper key.

Yesterday, some of you have tried a couple of non-traditional ways to hide the meaning of a text during our *CERC Cipher Contest*. Today, instead of the Cipher Contest, we have prepared a set of *Contest Ciphers* and other cryptography-related problems you are to solve algorithmically.

Your programs can be written in C, C++, or Java programming languages. The choice is yours but you will be fully responsible for the correctness and efficiency of your solutions. We need the correct answer produced in some appropriate time. Nothing else matters. You may choose any algorithm and any programming style.

All programs will read one single text file from the standard input. The results will be written to the standard output. Input and output formats are described in problem statements and must be strictly followed. Each text line (including the last one) should be always terminated by a newline character ("`\n`"), which is not considered a part of that line.

You are not allowed to use any other files, communicate over network, create threads or processes, or do anything else that could jeopardize the competition.

Happy enciphering and deciphering!

This problem set consists of 11 sheets of paper (including this one) and it contains 10 problems. Please make sure you have the complete set.

Vigenère Cipher Analysis

`analyse.c`, `analyse.C`, `analyse.java`

In this problem set, there is another problem (`vigenere`) asking you to implement the *Vigenère Cipher* encryption algorithm. This time, we will demonstrate one of the caveats of that cipher.

A secret organization Amateur Codebreakers Movement has a strong suspicion that bank robbers are planning another strike soon. Unfortunately, we do not know neither the name of the bank nor the exact day and time. ACM is able to eavesdrop the communication between robbers and their driver but the communication is encrypted using the Vigenère Cipher.

Your task is to try to break the cipher. You are given two words that are likely to appear in the original plaintext — so-called *cribs* (such words played an important role, for example, in breaking the famous Enigma code).

Input Specification

(For the specification of the Vigenère cipher, please refer to the `vigenere` problem.)

The input contains several instances. Each instance consists of four lines — the first line is an integer number K, $1 \leq K \leq 100$, the maximum length of the encryption key to be considered. The second and third lines contain the cribs W_1 and W_2, $1 \leq K \leq length(W_i) \leq 100$. The fourth line is the ciphertext C, $1 \leq length(C) \leq 100\,000$. Both the cribs W_1, W_2 and the ciphertext C consist only of uppercase letters of the standard English alphabet $\{A, B, C, ..., Z\}$. The input is terminated by a line containing one zero.

Output Specification

Your program must determine how many different plaintexts there exist that contain *both* of the given cribs *simultaneously* inside the same message and that will result into the given ciphertext using the Vigenère Cipher with some key Q, $1 \leq length(Q) \leq K$.

Print one line for each input instance:

- If there is exactly one plaintext satisfying all conditions, output that plaintext with no additional spaces.
- If two or more such plaintexts exist, print the word "`ambiguous`".
- If there is no such plaintext, print "`impossible`".

Sample Input

```
4
BANK
MONEY
FTAGUAVMKILCKPRIJCHRJZIYUAXFNBSLNNXMVDVPXLERWDSL
5
SECOND
PARSEC
SUKCTZHYYES
3
ACM
IBM
JDNCOFBEN
4
ABCD
EFGH
OPQRHKLMN
0
```

Output for Sample Input

```
WEWILLROBTHEBANKANDTAKEALLTHEMONEYTOMORROWATNOON
impossible
ambiguous
EFGHXABCD
```

Boring Card Game

`cards.c`, `cards.C`, `cards.java`

The Neal Stephenson's novel Cryptonomicon includes a cryptographic algorithm based on a deck of playing cards. It is emphasized that a proper shuffling is crucial for the cipher security. In this problem, we want to demonstrate the importance of randomness in cryptography by describing a card game that does not employ shuffling and is therefore foreseeable.

The game is a modification of poker, where not only all the cards are visible to everyone, but the players have no influence on the course of the game at all. Pretty boring, isn't it?

A *game session* is composed of (possibly) many games and is played by N players. For simplicity, we will assume that the players are sitting in a row and are numbered $1 \ldots N$ from left to right. The deck contains exactly $5 \times N$ cards numbered $1, 2, \ldots, 5N$.

At the beginning of each game, cards are dealt in three dealing rounds. First, two cards are dealt to each player in the left-to-right order. That is, player 1 gets the two top-most cards, then player 2 gets the next 2 cards and so on until the last player N gets his/her cards. In the second round, this step is repeated and every player gets another two cards in the same manner. Finally, each player gets one more card.

The player who receives all the five cards with the smallest numbers $(1, 2, 3, 4, 5$, not necessarily in this order) is the winner of the whole game session.

If nobody wins, the cards are collected and a new game is started. The cards are collected from the players from right to left and the cards of one player are always collected one-by-one in the reverse order then they were dealt. Each card is placed on top of the deck, another card onto it, and so on. That is, the top of the deck will contain cards of player number 1 and the six top-most cards will be the cards at positions 1, 2, $2N+1$, $2N+2$, $4N+1$, 3 in the original deck.

For example, in the game of two players the initial deck contains ten cards: A, B, C, D, E, F, G, H, I, J. In the first dealing round, player 1 gets the cards A and B, player 2 gets C and D. Then E and F is dealt to player 1, G and H to player 2, then I to player 1, and finally J to player 2. When collecting, the cards of the player 2 go first in the order J, H, G, D, C. Then we continue with player 1's cards I, F, E, B, and A. Since the cards are put onto the deck bottom-top, the final order of the cards after one game is A, B, E, F, I, C, D, G, H, and J.

Write a program that will determine the outcome of a game session so that you can spoil the game to its players.

Input Specification

The input contains several game sessions. Each session is described by two lines. The first line contains the number N, $1 \leq N \leq 1000$. The second line contains the card numbers $1 \ldots 5N$ in the order from the top of the deck to the bottom. Every two consecutive numbers on this line are separated by a single space. Each number will occur exactly once on that line.

The last decription is followed by a line containing one zero.

Output Specification

For each game session, output exactly one line. If no player ever wins, print "`Neverending game.`", otherwise output the sentence "`Player` P `wins game number` G`.`", where P is the player number and G is the number of the first game won (the first game is numbered 1). Please note that the result may exceed 2^{32} but it will always be less than 2^{63}.

Sample Input

```
2
2 3 9 7 4 8 5 1 10 6
2
2 6 9 7 4 8 5 1 10 3
5
16 12 18 11 20 15 19 24 8 6 25 1 7 22 14 2 3 10 13 17 4 5 21 9 23
0
```

Output for Sample Input

```
Player 1 wins game number 3.
Neverending game.
Player 2 wins game number 153.
```

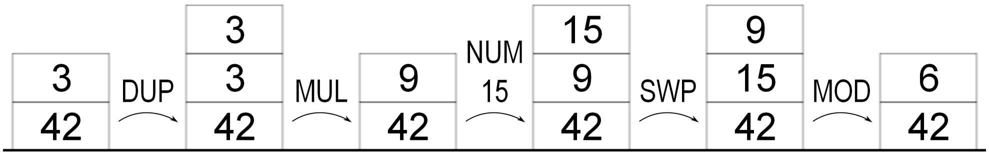
Stack Machine Executor

`execute.c`, `execute.C`, `execute.java`

Many ciphers can be computed much faster using various machines and automata. In this problem, we will focus on one particular type of machines called *stack machine*. Its name comes from the fact that the machine operates with the well-known data structure — *stack*. The later-stored values are on the top, older values at the bottom. Machine instructions typically manipulate the top of the stack only.

Our stack machine is relatively simple: It works with integer numbers only, it has no storage beside the stack (no registers etc.) and no special input or output devices. The set of instructions is as follows:

- **NUM *X***, where X is a non-negative integer number, $0 \leq X \leq 10^9$. The NUM instruction stores the number X on top of the stack. It is the only parametrized instruction.
- **POP**: removes the top number from the stack.
- **INV**: changes the sign of the top-most number. ($42 \rightarrow -42$)
- **DUP**: duplicates the top-most number on the stack.
- **SWP**: swaps (exchanges) the position of two top-most numbers.
- **ADD**: adds two numbers on the top of the stack.
- **SUB**: subtracts the top-most number from the "second one" (the one below).
- **MUL**: multiplies two numbers on the top of the stack.
- **DIV**: integer divison of two numbers on the top. The top-most number becomes divisor, the one below dividend. The quotient will be stored as the result.
- **MOD**: modulo operation. The operands are the same as for the division but the remainder is stored as the result.



All binary operations consider the top-most number to be the "right" operand, the second number the "left" one. All of them remove both operands from the stack and place the result on top in place of the original numbers.

If there are not enough numbers on the stack for an instruction (one or two), the execution of such an instruction will result into a program *failure*. A failure also occurs if a divisor becomes zero (for DIV or MOD) or if the result of any operation should be more than 10^9 in absolute value. This means that the machine only operates with numbers between $-1\,000\,000\,000$ and $1\,000\,000\,000$, inclusive.

To avoid ambiguities while working with negative divisors and remainders: If some operand of a division operation is negative, the absolute value of the result should always be computed with absolute values of operands, and the sign is determined as follows: The quotient is negative if (and only if) exactly one of the operands is negative. The remainder has the same sign as the dividend. Thus, 13 div $-4 = -3$, -13 mod $4 = -1$, -13 mod $-4 = -1$, etc.

If a failure occurs for any reason, the machine stops the execution of the current program and no other instructions are evaluated in that program run.

Input Specification

The input contains description of several machines. Each machine is described by two parts: the program and the input section.

The program is given by a series of instructions, one per line. Every instruction is given by three uppercase letters and there must not be any other characters. The only exception is the NUM instruction, which has exactly one space after the three letters followed by a non-negative integer number between 0 and 10^9. The only allowed instructions are those defined above. Each program is terminated by a line containing the word “`END`” (and nothing else).

The input section starts with an integer N ($0 \le N \le 10\,000$), the number of program executions. The next N lines contain one number each, specifying an input value V_i, $0 \le V_i \le 10^9$. The program should be executed once for each of these values independently, every execution starting with the stack containing one number — the input value V_i.

There is one empty line at the and of each machine description. The last machine is followed by a line containing the word “`QUIT`”. No program will contain more than 100 000 instructions and no program requires more than 1 000 numbers on the stack in any moment during its execution.

Output Specification

For each input value, print one line containing the output value for the corresponding execution, i.e., the one number that will be on the stack after the program executes with the initial stack containing only the input number.

If there is a program failure during the execution or if the stack size is incorrect at the end of the run (either empty or there are more numbers than one), print the word “`ERROR`” instead.

Print one empty line after each machine, including the last one.

Sample Input

```
DUP
MUL
NUM 2
ADD
END
3
1
10
50

NUM 1
NUM 1
ADD
END
2
42
43
```

```
NUM 600000000
ADD
END
3
0
600000000
1

QUIT
```

Output for Sample Input

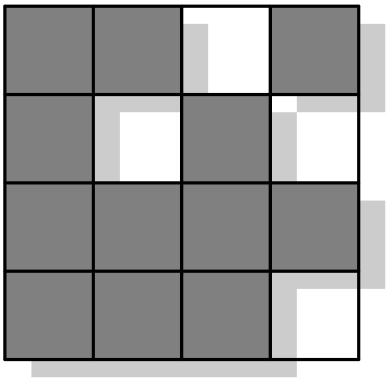
```
3
102
2502

ERROR
ERROR

600000000
ERROR
600000001
```

The Grille

`grille.c`, `grille.C`, `grille.java`

In the 16th century, there were no computers as we have today. On the other hand, even at that time it was necessary to protect messages from being read by inappropriate people. Old methods, like shaving a head of a slave, writing the message on his head, waiting until his hair grows back and then sending him through the area full of enemies, might work well but they took a long time. That is why new methods had to be invented. For example, an Italian mathematician Girolamo Cardano was the first one who described the *Grille* cipher.

For both enciphering and deciphering, you need a tool (basically, it is the cipher key) called the "grille". It is necessary that both parties (Alice and Bob) have the same grille. The grille looks like a rectangular grid of $N \times N$ unit squares with some of the squares being solid and some cut out to form "holes".



Let's have a grille with m holes. For enciphering, we write the first m letters of a message into the holes (starting in the first row from left to right, then continuing with other rows). Then we rotate the grille 90 degrees *clockwise* and write another m letters into the holes (again, from left to right and top to bottom). After that, we rotate the grille another 90 degrees and write another m letters. Then we repeat the same for the last time. At the end, we fill remaining places (if there are any) with random letters to make the ciphertext more secure. Please note that it is the *grille* that gets rotated, not the message!

For deciphering, we basically use the same algorithm, we just read the letters instead of writing them.

Input Specification

The input contains several test cases. Each test case contains description of a grille and a ciphertext. Your task is to decipher the message and write the plaintext to output.

Each test case starts with a line containing number N ($1 \leq N \leq 1000$), where N is the size of the grille. Then there are N lines containing the grille description. Each of those lines contains exactly N characters which are either the "hash" character "`#`" (solid/opaque material) or the uppercase letter "`O`" (hole).

Note: In praxis, the grille holes would be arranged in such a way that no position of the ciphertext is used more than once. In our problem, this is not guaranteed. Some grilles may contain holes that match the same position/letter of the ciphertext (after rotations). However, the deciphering algorithm is still the same.

After the grille description, there are another N lines with the enciphered message. Each of them contains exactly N characters — uppercase letters of alphabet.

The last test case is followed by a line containing one zero.

Output Specification

For each test case, output the deciphered message (plaintext) on one line with no spaces.

Sample Input

```
4
##O#
#O#O
####
###O
ARAO
PCEM
LEEN
TURC
3
O#O
###
O#O
ABC
DEF
GHI
0
```

Output for Sample Input

```
ACMCENTRALEUROPE
ACGIACGIACGIACGI
```

Stack Machine Programmer

`program.c`, `program.C`, `program.java`

Many ciphers can be computed much faster using various machines and automata. The trouble with such machines is that someone has to write programs for them. Just imagine, how easy it would be if we could write a program that would be able to write another programs. In this contest problem, we will (for a while) ignore the fact that such a "universal program" is not possible. And also another fact that most of us would lose our jobs if it existed.

Your task is to write a program that will automatically generate programs for the stack machine defined in problem `execute`.

Input Specification

The input contains several test cases. Each test case starts with an integer number N ($1 \leq N \leq 5$), specifying the number of inputs your program will process. The next N lines contain two integer numbers each, V_i and R_i. V_i ($0 \leq V_i \leq 10$) is the input value and R_i ($0 \leq R_i \leq 20$) is the required output for that input value. All input values will be distinct.

Each test case is followed by one empty line. The input is terminated by a line containing one zero in place of the number of inputs.

Output Specification

For each test case, generate any program that produces the correct output values for *all* of the inputs. It means, if the program is executed with the stack initially containing only the input value V_i, after its successful execution, the stack must contain one single value R_i.

Your program must strictly satisfy all conditions described in the specification of the problem `execute`, including the precise formatting, amount of whitespace, maximal program length, limit on numbers, stack size, and so on. Of course, the program must not generate a failure.

Print one empty line after each program, including the last one.

Sample Input

```
3
1 3
2 6
3 11

1
1 1

2
2 4
10 1

0
```

Output for Sample Input

```
DUP
MUL
NUM 2
ADD
END

END

NUM 3
MOD
DUP
MUL
END
```

Strange Regulations

`regulate.c`, `regulate.C`, `regulate.java`

Thank to cryptography, we are able to encrypt messages such that noone (except the intended recipient) is able to read them. However, encrypted messages are of no use if they do not actually *reach* the recipient. These days, computer network is the most typical mean to send such messages. In this problem, we will study the issues the networking providers have to solve. And remember: since the message is encrypted, we do not need to care about the network privacy anymore.

The network cables joining computers (servers) belong to different companies. A new anti-monopoly legislation prevents any company from owning more than two cables from each server. Furthermore, to avoid wasting resources, there is also a law specifying that the cable system owned by any single company cannot be redundant, i.e., removal of *any* of the cables will disconnect some two previously connected servers. Since the companies buy and sell the cables all the time, it is quite difficult to enforce these regulations. Your task is to write a program that does so.

Input Specification

The input contains several instances. The first line of each instance contains four integers N, M, C and T separated by spaces — the number of servers ($1 \leq N \leq 8\,000$), the number of cables ($0 \leq M \leq 100\,000$), the number of companies ($1 \leq C \leq 100$), and the number of cable-selling transactions ($0 \leq T \leq 100\,000$), respectively.

The following M lines describe the cables. Each of them contains three integers S_{j1}, S_{j2} and K_j, separated by spaces, giving the numbers of the servers S_{j1} and S_{j2} ($1 \leq S_{j1} < S_{j2} \leq n$) joined by that cable and the number of the company K_j ($1 \leq K_j \leq C$) initially owning the cable. For each pair of servers, there is at most one cable joining them. The initial state satisfies the regulations, i.e., each company owns at most two cables incident with each server, and the system of cables owned by a single company has no cycles.

Finally, each of the next T lines contains integers S_{i1}, S_{i2} and K_i describing one transaction in which the company K_i ($1 \leq K_i \leq C$) tries to buy a cable between servers S_{i1} and S_{i2} ($1 \leq S_{i1} < S_{i2} \leq N$).

The last instance is followed by a line containing four zeros.

Output Specification

For each input instance, output T lines describing the outcome of the transactions. The possible outcomes are

- "`No such cable.`" if the pair of servers is not joined by a cable,
- "`Already owned.`" if the cable is already owned by the company K_i,
- "`Forbidden:  monopoly.`" if the company K_i already owns two cables at S_{i1} or S_{i2},
- "`Forbidden:  redundant.`" if K_i owns at most one cable at each of S_{i1} and S_{i2}, but granting the ownership would create a cycle of cables owned by K_i,
- "`Sold.`" if none of the above restrictions apply. In this case, the ownership of the cable between S_{i1} and S_{i2} changes to K_i for the purpose of further transactions.

  Print one empty line after each instance.

Sample Input

```
4 5 3 5
1 2 1
2 3 1
3 4 2
1 4 2
1 3 3
1 2 3
1 2 3
1 4 3
2 3 3
2 4 3
2 1 1 1
1 2 1
1 2 1
0 0 0 0
```

Output for Sample Input

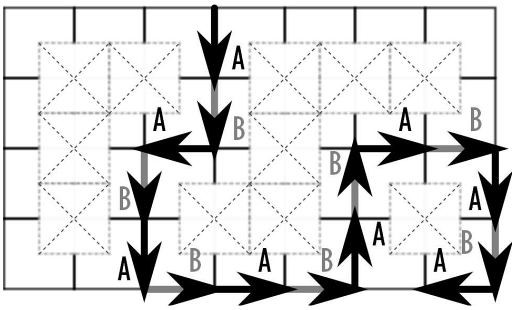
```
Sold.
Already owned.
Forbidden: monopoly.
Forbidden: redundant.
No such cable.

Already owned.
```

Racing Car Trail

`trail.c`, `trail.C`, `trail.java`

Have you ever read any description of some encryption algorithm? These descriptions almost always include messages being sent between *Alice* and *Bob*. We (the people organizing the 2011 Central Europe Regional Contest) think that those descriptions are too impersonal — considering these two people are probably the most famous cryptographers in the whole world, we know so little about them. They deserve more attention, don't you think? We can learn about their hobbies, for instance.

In their free time, Alice and Bob like to play a game inspired by *Tron*. In this game, you race a car through a square grid and you need to avoid hitting obstacles placed in the grid. Furthermore, the car leaves a permanent trail, which you also need to avoid. The car only moves in the four cardinal directions (east, west, north, or south). In their version of the game, Alice and Bob alternate in controling the car—Alice starts, moves the car from its initial position to one of the adjacent positions in the grid, then Bob takes over and moves the same car to another adjacent position, etc.



The player who crashes the car (i.e., moves it to a position occupied by an obstacle, or to one of the previously visited positions) loses. Both Alice and Bob are incredibly skilled players and never make mistakes; in particular, they only crash if there is no possible move from their current position that would avoid it. Given the map of the obstacles, your task is to determine which player wins from which initial position.

Input Specification

The input contains descriptions of several game fields. The first line of each description contains two integers N and E ($1 \leq N, E \leq 100$) — the size of the grid in the north-south and in the east-west directions. The following N lines describe the map. Each of the lines contains a string of E characters, where the j-th character on the i-th line determines the state of the position with coordinates (j, i). The possible characters are "`.`" (a dot) if the position is empty and the uppercase letter "`X`" if there is an obstacle. All positions not covered by the map (i.e., with coordinates (j, i) such that $i \leq 0$ or $j \leq 0$ or $i > N$ or $j > E$) are forbidden and not used in the game, they work as if there were obstacles.

The last game field is followed by a line containing two zeros.

Output Specification

For each game field, output N lines of strings of length E, showing whether Alice or Bob wins when the game starts from the given location. The j-th character on the i-th line should be "`A`" if Alice wins when starting from the position (j, i), "`B`" if Bob wins, or "`X`" if the position contains an obstacle.

After each output, print one empty line.

Sample Input

```
1 1
.
3 3
...
.X.
...
1 4
....
3 3
X.X
...
X.X
5 8
........
.XX.XXX.
.X..X...
.X.XX.X.
........
0 0
```

Output for Sample Input

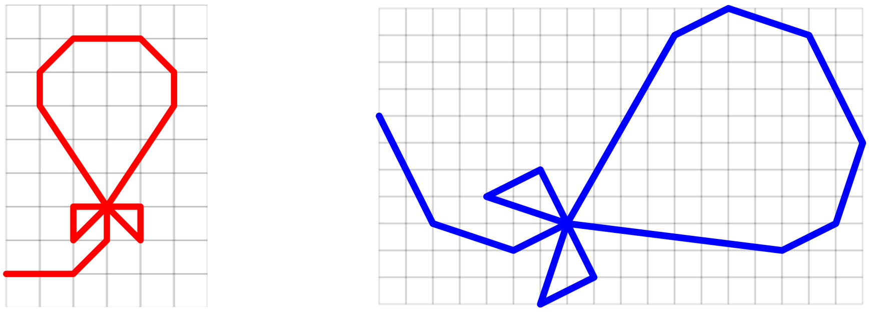
```
B

AAA
AXA
AAA

AAAA

XBX
BAB
XBX

BABABABA
AXXBXXXB
BXBAXABA
AXAXXBXB
BABABABA
```

Unchanged Picture

`unchange.c`, `unchange.C`, `unchange.java`

Steganography is a special way to protect messages — instead of encryption, the message is somehow *hidden*. Historically, it was quite a popular technique, but nowadays it is superseded by ciphers, especially those based on keys. However, sometimes it may still be in use. One of the digital steganographic techniques is to hide small pieces of information into a digital image. The image modification is so small that it cannot be spotted by a human eye, but the information (such as a text message) is there and readable by computers. Your task is to compare two images and find such (possibly small) differences.

In this problem, we will focus on vector pictures. Your program is given two pictures and it should decide whether they contain the same image. Geometrically speaking, decide whether the two picture are *similar*, that means whether they can be transformed into each other using translation, rotation, and uniform scaling (but not mirroring). An example of two similar pictures follows.



Input Specification

The input file consists of several test cases, each of them containing two vector pictures. Each picture is described by a sequence of instructions for a plotter device, one instruction per row. Every instruction begins with an uppercase letter followed by one space character and two integer coordinates separated by another space. The letter is either "`L`" (draw a line) or "`M`" (move without drawing). The coordinates specify the place to which the line is to be drawn or the current position moved. Coordinates are always given relatively to the end position of the previous instruction. The first instruction is relative to some (unspecified) starting point.

The last instruction of each picture is followed by a row containing the letter "`E`" (end) and an empty line. The last test case will be followed by a row containing the letter "`Q`" (quit).

The number of instructions for any picture is between 0 and 1000, inclusive. No instruction has both coordinates equal to zero. The absolute value of all (relative) coordinates is at most 1000.

Output Specification

For each test case, print one line containing either the word "`YES`" (two pictures are similar) or "`NO`" (pictures are not similar).

Sample Input

```
L 3 1
L 3 1
M 3 0
M 0 1
L -3 -1
E

L 1 0
L -1 0
E

L 1 0
L 0 1
E

L 1 0
L -1 -1
E

L 2 0
L 1 1
L 0 1
L 2 3
L 0 1
L -1 1
L -2 0
L -1 -1
L 0 -1
L 2 -3
M 1 0
L -2 0
M 2 0
L 0 -1
L -1 1
L -1 -1
L 0 1
E

L 2 -4
L 3 -1
L 2 1
L 1 -2
L -2 -1
L 1 3
L -3 1
L 2 1
L 1 -2
L 8 -1
L 2 1
L 1 3
L -2 4
L -3 1
L -2 -1
L -4 -7
E

Q
```

Output for Sample Input

```
YES
NO
YES
```

Unique Encryption Keys

`unique.c`, `unique.C`, `unique.java`

The security of many ciphers strongly depends on the fact that the keys are unique and never re-used. This may be vitally important, since a relatively strong cipher may be broken if the same key is used to encrypt several different messages.

In this problem, we will try to detect repeating (duplicate) usage of keys. Given a sequence of keys used to encrypt messages, your task is to determine what keys have been used repeatedly in some specified period.

Input Specification

The input contains several cipher descriptions. Each description starts with one line containing two integer numbers M and Q separated by a space. M ($1 \leq M \leq 1\,000\,000$) is the number of encrypted messages, Q is the number of queries ($0 \leq Q \leq 1\,000\,000$).

Each of the following M lines contains one number K_i ($0 \leq K_i \leq 2^{30}$) specifying the identifier of a key used to encrypt the i-th message. The next Q lines then contain one query each. Each query is specified by two integer numbers B_j and E_j, $1 \leq B_j \leq E_j \leq M$, giving the interval of messages we want to check.

There is one empty line after each description. The input is terminated by a line containing two zeros in place of the numbers M and Q.

Output Specification

For each query, print one line of output. The line should contain the string "`OK`" if all keys used to encrypt messages between B_j and E_j (inclusive) are mutually different (that means, they have different identifiers). If some of the keys have been used repeatedly, print one identifier of *any* such key.

Print one empty line after each cipher description.

Sample Input

```
10 5
3
2
3
4
9
7
3
8
4
1
1 3
2 6
4 10
3 7
2 6

5 2
1
2
3
1
2
2 4
1 5

0 0
```

Output for Sample Input

```
3
OK
4
3
OK

OK
1
```

Vigenère Cipher Encryption

`vigenere.c`, `vigenere.C`, `vigenere.java`

One of the oldest and most common encryption algorithms is *Vigenère Cipher*. It is quite an old thing — a similar encryption was first described in 1553 by Giovan Battista Bellaso and improved in 1586 by Blaise de Vigenère.

Vigenère encryption produces a single letter of ciphertext for each letter of plaintext, combining one plaintext letter with one single letter of a *key* on the corresponding position. If the key is shorter than the plaintext, it is simply repeated as needed, e.g. for a key of length 3 and plaintext of length 7, letters will be combined like this (K_i is the key letter, P_i is the plaintext letter, and C_i is the resulting ciphertext letter).

$$\begin{array}{ccccccc} K_1 & K_2 & K_3 & K_1 & K_2 & K_3 & K_1 \\ P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 \\ \hline C_1 & C_2 & C_3 & C_4 & C_5 & C_6 & C_7 \end{array}$$

The letter of the key specifies how many positions should be the plaintext letter "shifted forward" in the alphabet. If a key letter is A, the correspoding plaintext letter will be shifted by one character, B means two positions, etc. The alphabet is considered circular, so if the last letter (Z) should be shifted, it becomes A again. Please note that A (key) combined with another A (plaintext) will result in B, which may be a little unusual for the common Vigenère cipher. The Vigenère square at the end of this problem statement gives an overview how letters of a plaintext get combined with letters of a key to produce the ciphertext.

Your task is to write a program that will encrypt messages using the Vigenère cipher with a given key.

Input Specification

The input contains several instances. Each instance consists of two lines, the first line is the encryption key and the second line is the plaintext. Both key and plaintext consist of uppercase letters of the English alphabet $\{A, B, C, ..., Z\}$. The length of the key will be between 1 and 1000, the length of the plaintext between 1 and 100 000, inclusive.

Input is terminated by a line containing one zero.

Output Specification

For each input instance, output the ciphertext — the encrypted version of the message.

Sample Input

```
ICPC
THISISSECRETMESSAGE
ACM
CENTRALEUROPEPROGRAMMINGCONTEST
LONGKEY
CERC
0
```

Output for Sample Input

```
CKYVRVIHLUUWVHIVJJU
DHAUUNMHHSRCFSEPJEBPZJQTDRAUHFU
OTFJ
```

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Vigenère square:
Mapping a given plaintext letter (column) and a key letter (row) to the resulting ciphertext letter