Day 1 Question 1: Spam

Your solution: N :\spam\spam.{pas, c, cpp}
Input file: spam.in
Output file: spam.out

Unsolicited email (spam) is annoying and clutters your mailbox. You are to write a *spam filter* - a program that reads email messages of regular ASCII characters and tries to determine whether or not each message is spam.

How can we determine whether or not a message is spam? Spam contains words and phrases that are not common in genuine email messages. For example, the phrase

MAKE MONEY FAST, HONEY!!

is in all-uppercase, contains the word "money" and ends with a double exclamation mark.

One way to create a spam filter is to read through many spam and non-spam messages and to come up with a set of rules that will classify any particular message as spam or not. This process can be tedious and error prone to do manually. Instead you will write a program to automate the process.

A useful step in automatic classification is to split the text up into set of *trigrams*. A trigram is a sequence of three adjacent characters that appear in the message. A trigram is case sensitive. The example above is composed of the trigrams:

```
MAK
AKE
KE
E M
MO
MON
ONE
NEY
EY
Y F
FA
FAS
AST
ST,
T,
, H
HO
HON
ONE
NEY
EY!
Y!!
```

If we examine a sample of spam and non-spam messages we find that some trigrams are more common in spam; whereas others are more common in non-spam. This observation leads to a classification method:

- Examine a sample consisting of a large number of spam messages. Count the number of times that each trigram occurs. In the example above, there are 20 distinct trigrams; the trigrams ONE and NEY occur twice each and the remaining 18 trigrams occur once each. (Trigrams that do not occur are considered to occur 0 times.) More formally, for each trigram *t* we compute the frequency $f_{spam}(t)$ with which it occurs in the sample of spam.

- Examine a sample consisting of a large number of non-spam messages. Compute $f_{non\text{-}spam}(t)$, the frequency with which each trigram *t* appears in the sample of non-spam.
- For a each message to be filtered, compute $f_{message}(t)$ for each trigram *t*.
- If $f_{message}$ resembles f_{spam} more closely than it resembles $f_{non\text{-}spam}$ it is determined to be spam; otherwise it is determined to be non-spam.
- A *similarity measure* determines how closely f_1 and f_2 resemble one another. One of the simplest measures is the cosine measure:

$$\text{similarity}\ (f_1, f_2) = \frac{\sum_t f_1(t) \cdot f_2(t)}{\sqrt{\sum_t [f_1(t)]^2} \cdot \sqrt{\sum_t [f_2(t)]^2}}$$

  Then we say that a message is spam if

  similarity($f_{message}$, f_{spam}) > similarity($f_{message}$, $f_{non\text{-}spam}$)

The first line of input contains three integers: *s* the number of sample spam messages to follow; *n* the number of sample non-spam messages to follow; *c* the number of messages to be classified as spam or non-spam, based on trigram the trigram frequencies of the sample messages. Each message consists of several lines of text and is terminated by a line containing "`ENDMESSAGE`". This line will not appear elsewhere in the input, and is not considered part of the message.

For each of the *c* messages, your program will output two lines. On the first line, output similarity($f_{message}$, f_{spam}) and similarity($f_{message}$, $f_{non\text{-}spam}$). On the second line print the classification of the message ("spam" or "non-spam"). Round the numbers to five decimal digits.

When forming trigrams, we never include a newline character. We don't include trigrams that span multiple lines, either. So in the first spam message of Sample Input 1, the only trigrams are:

```
"AAA", "BBB", "BB ", "B  ", "  C", " CC", and "CCC".
```

Sample Input 1

```
2 1 1
AAAA
BBBB   CCCC
ENDMESSAGE
BBBB
ENDMESSAGE
AAAABBBB
ENDMESSAGE
AAABB
ENDMESSAGE
```

Output for Sample Input 1

```
0.21822 0.73030
non-spam
```

Sample Input 2

Found in the file d1q1sample.txt

Output for Sample Input 2

```
0.28761 0.20595
spam
0.44314 0.49243
non-spam
```

Day 1 Question 2: Game Show Math

Your solution: N:\math\math.{pas, c, cpp}
Input file: math.in
Output file: math.out

A game show in Britian has a segment where it gives its contestants a sequence of positive numbers and a target number. The contestant must make a mathematical expression using all of the numbers in the sequence and only the operators: +, -, *, and, /. Each number in the sequence must be used exactly once, but each operator may be used zero to many times. The expression should be read from left to right, without regard for order of operations, to calculate the target number. It is possible that no expression can generate the target number. It is possible that many expressions can generate the target number.

There are three restrictions on the composition of the mathematical expression:

- the numbers in the expression must appear in the same order as they appear in the input file
- since the target will always be an integer value (a positive number), you are only allowed to use / in the expression when the result will give a remainder of zero.
- you are only allowed to use an operator in the expression, if its result is an integer from –32000 .. 32000.

Input

The input file describes multiple test cases. The first line contains the number of test cases *n*.

Each subsequent line contains the number of postive numbers in the sequence *p*, followed by *p* positive numbers, followed by the target number. Note that $0 < p \leq 100$. There may be duplicate numbers in the sequence.

Output

The output file should contain an expression, including all *k* numbers and (*k*-1) operators plus the equals sign and the target. Do not include spaces in your expression. Remember that order of operations does not apply here. If there is no expression possible output "NO EXPRESSION".

Sample Input

```
3
3 5 7 4 3
2 1 1 2000
5 12 2 5 1 2 4
```

Output for Sample Input

```
5+7/4=3
NO EXPRESSION
12-2/5*1*2=4
```

Day 1 Question 3: Return to Blind Man’ s Bluff

Your solution: N:\bluff\bluff.{pas, c, cpp}
Input file: bluff.in
Output file: bluff.out

You have all played Blind Man’ s Bluff before, in an earlier Stage of life. In this sequel, you have even less information to work with, and you need to provide even more detailed responses. Recall that the "game" of Blind Man’ s Bluff is as follows: the player is placed in a known *n* x *m* playing area (with obstacles of size 1 square unit placed at various points in the playing area) pointing in one of the 4 compass directions (North, East, South, West). After moving in forward (F), turning right (R), or turning left (L), you were to determine your original starting position and the compass direction you were pointing in (either N=North, S=South, E=East, W=West).

We augment this game slightly now. Instead of simply determining your starting position and your direction, you are to determine the game board layout. That is, you are *really* playing this game blind. To help you in this endeavor, you are given the dimensions of the board, your starting position (in co-ordinates) and your starting orientation (either N=North, S=South, E=East, W=West).

However, you have access to a query engine that will tell you whether the move you wish to perform is possible. It is always possible to turn right or left, though it may be impossible to move forward if there is an obstacle or wall (i.e., the boundary of the board) in front of you. This query engine is contained within module called `query` that has the following functions:

- `procedure Right()`
  turn right 90 degrees in the playing area
- `procedure Left()`
  turn left 90 degrees in the playing area
- `function Forward() : integer`
  move forward and return 1 if there is no obstacle or wall in the position immediately (1 unit) in front of you. Returns 0 and does not change position or compass orientation if there is a wall or an obstacle 1 unit in front of you.

You may assume that it is possible to map out the entire game area. For example, you will **NOT** have the following type of playing area in the 5x5 case

```
. . . . X
. X X . .
. X . X .
. . X X .
. . . . .
```

since if you were placed at position (3,3), you would be unable to determine the obstacles at positions (1,5), (2,2) and (4,4). In other words, there are no unreachable spaces in the playing area.

Testing Your Program

The query module reads the description of a test case from the file `query.in`. This file contains the values *n, m, r,* and *c*, each on a separate line, followed by the game board in the format described above. You can modify this file by hand in order to test your program. Of course, your program must not read directly from this file.

The query module outputs a file named `query.log` which describes the calls that your program made to the module.

Instructions for Pascal Programmers

To access the library, your program must contain the statement

```
uses query;
```

For your information, here are the function and procedure declarations:

```
procedure Right;
procedure Left;
function Forward:integer;
```

Instructions for C/C++ Programmers

Use the instruction `#include "query.h"` in your source code. Use the "Open Project..." command to create a new project. Add your source file and the module `query.o` to your project.

Input

Input will be on 5 separate lines. The first line contains *n* (number of rows). The second line contains *m* (number of columns). Following that, you are given your starting row *r* (1 <= *r* <= *n*) on one line, followed by your starting column position *c* (1 <= *c* <= *m*) on the next line. The fifth line will contain the starting orientation: that is, one of the characters, N, S, E, W.

Output

You are to output the game board on *n* separate lines, with *m* characters on each line. The m characters can be either . (to represent a non-obstacle position) or X (to represent an obstacle at that position). To disambiguate, you must draw the board so that the left side of the screen is west (and the right is east, and so on).

Sample Input

```
2
2
1
1
N
```

Sample Query Interaction

Forward	→ Returns false
Right	
Forward	→ Returns true
Forward	→ Returns false

Forward	→ Returns false
Right	
Forward	→ Returns true
Right	
Forward	→ Returns false

Output for Sample Input

```
. .
X .
```