CONTEST PROBLEMS Fri Apr 1 05:00:16 EDT 2016

Easier problems are first.

spoonerisms
A switch causes a stitch.
Boston Preliminary 2012

fractions
Simplify!
Harvard Selection Contest Fall 2015

fibonacci
Tail recursive fibonnaci.
Northeastern Preliminary at Harvard 1997.

drunkard
Walk the walk and compute.
Boston Preliminary 2004.

buffalo
Lost in space? Call out the hound!
Boston Preliminary 2008

lexemes
Words, words, but more than words.
Boston Preliminary 2010

snowluck
Snow Luck Maze analyzed.
Boston Preliminary 2001.

Spoonerisms

You have been asked to write a program that will produce simple Spoonerisms. The Reverend William Archibald Spooner (1844-1930) was reputed to have the tendency to switch consonants and vowels in words, and thus to turn sentences like

You have missed all my history lectures.

into

You have hissed all my mystery lectures.

Although the Reverend was prone to such slips of the tongue, most 'Spoonerisms' attributed to him were actually made up by the Reverend's students and colleagues for the fun of it.

You have been asked to write a program that will create simple Spoonerisms by taking a sequence of words and switching the consonant strings at the beginnings of the first and last word.

Input

For each of several test cases, one line containing two or more words. The words, which will consist solely of lower case letters (for simplicity), are separated by a single space character. No other characters are on the input line. No line will be longer than 80 characters.

Input ends with an end of file.

Output

For each test case, a copy of the test case input line with the consonant strings beginning the first and last words switched (see samples below). The longest strings of consonants at the beginnings of the two words should be switched. A 'y' should be treated as a consonant if it begins a word, and as a vowel otherwise (this is also for simplicity and is not a generally valid rule). It is possible that a string of consonants will be of zero length.

Sample Input

```
ease my tears
pack of lies
oiled bicycle
lighting a fire
dental receptionist
selling yaks
mystery house
```

Sample Output

```
tease my ears
lack of pies
boiled icycle
fighting a lire
rental deceptionist
yelling saks
hystery mouse
```

```
File:      spoonerisms.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Mon Oct  1 05:19:11 EDT 2012

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.
```

Fractions

You have been asked to write the numeric output routine for a calculator in which numbers such as 0.333333333 are output as 1/3 rather then 0.333, on the grounds that 1/3 is more accurate and sometimes more user friendly than 0.333.

In general a fractional representation has the form N/D where

- N and D are integers
- 0 < N < D
- N and D have no common factor (other than 1)

Your code is to be part of a calculator that allows inputs of the form N/D and thus often produces results that have the same general form, but the calculator uses double precision floating point numbers internally.

However, when the result is a fraction, it will be that faction to very high accuracy. Internally the accuracy is over 15 decimal digits. So we assume that if a number is the same as a fraction to within 9 decimal digits, it equals the fraction. This allows 6 decimal digits for inaccuracies of arithmetic, such as those that might arise from summing 1,000,000 fractions.

You are to write a program that determines whether a number between 0 and 1 is represented by a fraction with D <= 1000 within a precision of 9 decimal digits.

IMPORTANT: You do NOT need to optimize. The judge is interested in getting a good algorithm for this application, so he coded one brute force solution and two optimal ones, but the brute force solution ran only 4 times slower than either optimal solution. So its not easy to discriminate optimal from brute force solutions in a contest. Thus the judge's data set for this problem is small and a brute force solution suffices.

Input

For each of very many test cases, a line containing just the number n to be represented. 0 <= n < 1.

Input ends with an end of file.

Output

For each test case, a line of the form

```
n r
```

where r is either a fraction of the form N/D meeting the requirements given above (in particular, D <= 1000 and |n-N/D| < 0.0000000005 == 0.5e-9), or is just a '-' indicating that there is no such fraction. The input will be such that the results are unambiguous.

In the output lines, n should be printed with 15 decimal places.

```
Sample Input
------ -----

0.333333333333333
0.33333334
0.066666666666666
0.06666667
0.111111111111111
0.555555555555555
0.123123123123123
0.729729729729729
0.729729727000000
0.729729729700000


Sample Output
------ ------

0.333333333333333 1/3
0.333333340000000 -
0.066666666666666 1/15
0.066666670000000 -
0.111111111111111 1/9
0.555555555555555 5/9
0.123123123123123 41/333
0.729729729729729 27/37
0.729729727000000 -
0.729729729700000 27/37



File:      fractions.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Sun Sep 20 11:33:11 EDT 2015

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.
```

```
Tail Recursive Fibonacci
---- --------- ---------

The Fibonacci function is directly defined by the
rewrite rules:

    fibonacci ( 0 ) ===> 0
    fibonacci ( 1 ) ===> 1
    fibonacci ( n ) ===> fibonacci ( n - 2 )
                         +
                         fibonacci ( n - 1 )
                            if n >= 2
        where n >= 0 is an integer

A straightforward encoding of these rules as a recursive
function leads to an implementation whose running time
is exponential in the argument n.

There is a well known loop implementation whose running
time is linear in n.

Any loop can be encoded as a 'tail-recursive' function.
The rewrite rules to do so for the Fibonacci function
are:

    fibonacci ( 0 ) ===> 0
    fibonacci ( n ) ===> fibonacci_helper ( n, 1, 0, 1 )
                            if n >= 1
        where n >= 0 is an integer

    fibonacci_helper ( n, m, fib_m_minus_1, fib_m )
        ===> fib_m      if m == n
        ===> fibonacci_helper ( n, m + 1, fib_m,
                                fib_m_minus_1 + fib_m )
                        otherwise
        where m, n, fib_m_minus_1, and fib_m
              are integers

Here fib_m is the value of 'fibonacci ( m )' and
fib_m_minus_1 is the value of 'fibonacci ( m - 1 )'.
```

```
You know that fibonacci_helper implements a loop because
in its recursive case it calls itself ONLY as the last
thing it does.  Thus it is said to be 'tail-recursive'.

Compilers typically translate tail-recursive functions
into loops in order to save stack space.  This can be
done simply by replacing the tail-recursive call with
code that copies the tail-recursive call arguments to
the location of the current function execution's argu-
ments and jumps back to the beginning of the current
function.

Every loop can be readily re-written as a tail-recursive
function and every tail-recursive function can be
readily re-written as a loop.

You have been asked to implement the functions fibonacci
and fibonacci_helper according to the tail-recursive
second set of rewrite rules given above.  You are also
asked to print at trace of the execution of these
functions.


Input
-----

For each of several test cases, one line containing just
the integer n to be given to the fibonacci function.
0 <= n <= 40.  Note that fibonacci(40) fits in a 32 bit
signed integer.  Input ends with an end of file.


Output
------

For each test case, first print a single line containing
the integer n input to the test case (this is in effect
an exact copy of the test case input line).  Then call
fibonacci ( n ).
```

At the beginning of each of the functions fibonacci and fibonacci_helper you are to print one line that represents the call to the function. This line has the name of the function, and the arguments as integers separated by commas inside parentheses.

At the end of each of the functions fibonacci and fibonacci_helper you are to print one line that represents the return from the function. This line has the name of the function, followed by an equals sign '=', followed by the result returned by the function.

There must not be any space or tab characters in any printed line. There must not be any sign characters or leading zeros in the integers printed (zero prints as '0').

Notes:

After you add code to the end of fibonacci_helper to print the return part of the trace, this function will no longer be tail-recursive.

You might find it amusing to code your program's input loop as a tail-recursive main function.

Sample Input
------ -----

```
0
1
4
```

Sample Output
------ ------

```
0
fibonacci(0)
fibonacci=0
1
fibonacci(1)
fibonacci_helper(1,1,0,1)
fibonacci_helper=1
fibonacci=1
4
fibonacci(4)
fibonacci_helper(4,1,0,1)
fibonacci_helper(4,2,1,1)
fibonacci_helper(4,3,1,2)
fibonacci_helper(4,4,2,3)
fibonacci_helper=3
fibonacci_helper=3
fibonacci_helper=3
fibonacci_helper=3
fibonacci=3
```

File: fibonacci.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Sun Mar 20 05:22:38 EDT 2016

The 1D Drunkard

Some scientific algorithms require random numbers as input. However, with modern inexpensive computers, which do not have error detecting RAM memory, it is also important to be able to repeat computer runs, in order to check that they are correct.

A solution is to use a pseudo-random number generator that produces an apparently random but actually repeatable series of numbers.

The following is a classic pseudo-random number generator:

```
r(0) = seed     /* must not be zero */
r(i+1) =  r(i) * (7**5) mod (2**31 - 1)
```

where

```
7**5 = 16807
2**31 - 1 = 2147483647
0 < seed < 2147483647
```

Here r(0), r(1), r(2), ... is the sequence of pseudo-random numbers generated. Because 2**31 - 1 is prime, this sequence is 2**31 - 2 numbers long before it repeats. This particular sequence has been extensively tested and found to do very well in common tests of randomness.

You are asked to use this random number generator to simulate a drunkard's walk in a one dimensional world. The drunkard starts at position zero. A random number is acquired. If that is odd, the drunkard 'steps right' by adding 1 to his current position. If it is even, the drunkard 'steps left' by subtracting 1 from his current position. Successive steps are taken as successive random numbers are acquired. The first random number acquired is the seed, and thereafter the equation

```
next_number =  ( last_number * 16807 ) mod 2147483647
```

is used to produce more random numbers. The current position can become a negative integer.

Note

When programming this in C or C++ use the 'long long' number type, as in:

```
long long multiplier = 16807;
long long modulus = 2147483647;
int seed, next;
. . . .
next = seed;    // First random number.
. . . .
// Compute next random number.
next = (int)
       ( ( multiplier * next ) % modulus );
```

The JAVA code is the same but 'long long' is replaced by 'long'.

Input

Lines each of which contains one command. There are two kinds of command.

The W m seed

command, where m > 0 and seed > 0 are integers and W is the character ‘W’, causes the output of a graph of an m step drunken walk, with the first random number being seed.

The H m n seed

command, where m > 0, n > 0, and seed > 0 are integers, outputs a histogram of the position the drunkard ends up in after after m steps. The drunkard’s m-step walk is simulated n times, and H(p) is computed to be the number of those times that the drunkard’s final position after m steps is p. The random number is NOT reset after each walk simulation, so except for the first walk, the first random number of a walk is the next random number after the last random number of the previous walk. The first random number of the first walk is of course the seed. You can assume m <= 1000.

Input ends with an end of file.

Output

The first thing each command outputs is a line containing an exact copy of the input command line. This can be made by printing the command character and the input integers separated by single space characters.

Then the ‘W’ command outputs a graph consisting of m+1 lines, each outputting one position. The first position output is 0, and the next m lines output the position after each of the m steps. The line outputting a position p consists of exactly p + 35 space characters followed by a single ‘*’ character, and nothing else. The input will be such that the position never gets outside the range from -35 to +35 for a ‘W’ command.

The ‘H’ command outputs a histogram consisting of one line for each of the values p = -m, -m+2, -m+4, ..., m-4, m-2, m. This line contains

p H(p) P(p)

where p is the position, H(p) is the number of times the drunkard ended in position p after m steps starting in position 0, and P(p) is a theoretical estimate of H(p) computed by

$$P(p) = 2 * n * N(p,m)$$

$$N(p,m) = \exp (- p^{**}2 / (2 * m)) / \sqrt{ 2 * PI * m }$$

Here p and H(p) are integers, but P(p) is a floating point number. p, H(p), and P(p) must be each be printed right adjusted in 15 columns, and P(p) must have exactly 1 decimal place.

Note that for p equal -m+1, -m+3, ..., m-3, m-1, P(p) is zero, which is why no lines are printed for these p. If m is even p must be even, and if m is odd p must be odd, for the drunkard at an even position must step to an odd position, and at an odd position must step to an even position.

N(p,m) is the normal probability distribution with mean 0 and standard deviation sqrt(m). p can be shown to be a random variable with the same mean and standard deviation. The reason for the '2 *' in the equation for P(p) is that H(p) is zero for every other value of p, so H(p) is approximated by the integral of n * N(p,m) over an interval of length 2. Another way of putting this is that the sum of all the H(p) for different p is n, and to make the sum of the P(p) for p = -m, -m+2, ..., m-2, m be approximately n, we have to add the factor '2 *'.

Sample Input

```
------ -----

W 20 7456353
H 10 1000 276089259
```

Sample Output

```
------ ------

W 20 7456353
                                        *
                                       *
                                      *
                                     *
                                    *
                                   *
                                  *
                                 *
                                 *
                                *
                                *
                                *
                                *
                                *
                               *
                              *
                              *
                              *
                              *
                              *
                              *
                              *
H 10 1000 276089259
             -10              4            1.7
              -8              7           10.3
              -6             41           41.7
              -4            106          113.4
              -2            223          206.6
               0            254          252.3
               2            193          206.6
               4            120          113.4
               6             43           41.7
               8              9           10.3
              10              0            1.7
```

```
File:      drunkard.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Sun Mar 20 08:10:06 EDT 2016

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.
```

Buffalo Finder
------- ------

Ranchy Flatman is a buffalo rancher who has lived on the Plains for decades. He has a ranch on which he keeps buffalo in a field with three sides. Each side is bounded by a perfectly straight fence which actually extends in both directions well beyond Ranchy's field, and which serves as a boundary between various fields owned by various other people. In all there are 7 fields, Ranchy's triangular field, and 6 neighboring fields.

Sometimes Ranchy loses track of a buffalo. Then he sends his trusty Beagle Issy ('I Smell You') out to find the errant buffalo. Issy wears a GPS receiver and a radio, and this sends Issy's position back to Ranchy. When Issy finds the buffalo, she stops, and Ranchy then knows the buffalo's GPS coordinates. If the buffalo has gotten lost in Ranchy's field, Ranchy goes out to find the buffalo, but if the buffalo is in a neighbor's field, Ranchy must call up the neighbor who will go with Ranchy to retrieve the Buffalo and Issy.

In order to make this work, Ranchy's daughter has worked out the following naming system for fields, and programmed the family computer to tell Ranchy which field Issy is in. The corners of Ranchy's triangular field are given the names 1, 2, and 3 in clockwise order, and the fences are given names 12, 23, and 31 in clockwise order. A given field can be either to the left or right of a given fence. So we can give a field a name of the form

D12 D23 D31

where Dxy is 'L' if the field is to the left of fence xy and 'R' if the field is to the right of fence xy when traveling in the direction from x to y. Thus Ranchy's triangular field is named RRR and if you cross fence 12 from this field you enter field LRR.

To find out which field Issy is in, the GPS coordinates of Issy and the corners 1, 2, and 3 are used. The GPS coordinates are treated as integer coordinates of points in a flat plane.

Due to an unfortunate accident, Ranchy's daughter's computer program has been lost, and as she is off at college and in the middle of exams, you have been tasked to replace it.

Input

For each test case, one line of the form

x1 y1 x2 y2 x3 y3 xi yi

where (x1,y1), (x2,y2), (x3,y3) are the coordinates of the corners 1, 2, and 3, respectively, and (xi,yi) are Issy's coordinates. All coordinates are integers. Input ends with an end of file.

For simplicity, the input will be such that Issy is never exactly on a fence.

Output

For each test case one line containing just the name of the field containing Issy.

```
Sample Input
------ -----
-3 -3 0 6 3 -3 0 0
-3 -3 0 6 3 -3 -5 0
-3 -3 0 6 3 -3 0 10
-3 -3 0 6 3 -3 5 0
-3 -3 0 6 3 -3 10 -4
-3 -3 0 6 3 -3 0 -4
-3 -3 0 6 3 -3 -10 -4


Sample Output
------ ------

RRR
LRR
LLR
RLR
RLL
RRL
LRL


File:      buffalo.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Wed Oct 15 03:05:19 EDT 2008

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.
```

Lexemes

You have been asked to scan lines of input text into lexemes.

For example, given the input line

```
    x = 5*y + "hello world";
```

you are to output

```
    |x| |=| |5|*|y| |+| |"hello world"|;|
     s w o w n o s w o w qqqqqqqqqqqq p
```

The definitions are

```
    <lexeme>     ::= <symbol>
                   | <whitespace>
                   | <operator>
                   | <number>
                   | <quoted-string>
                   | <punctuation>
                   | <illegal>
    <symbol>     ::= <letter><letter-or-digit>*
    <whitespace> ::= <single-space-character>+
    <operator>   ::= '+' | '-' | '*' | '/' | '=' | '.'
    <number>     ::= <digit>+ <fraction-option>
    <fraction-option> ::= <empty> | '.' <digit>+
    <quoted-string>   ::=
        '"' <character-representative>* '"'
    <character-representative> ::=
        <character-except-"-or-\> | '\"' | '\\'
    <punctuation> ::= ',' | '(' | ')' | ';'

    <illegal> ::=
        <any-character-that-starts-no-other-lexeme>
```

Here <x>* means zero or more <x>'s, <x>+ means one or more <x>'s, and <empty> means the empty character string.

Given a position in the input, the next lexeme is the LONGEST lexeme that can be found starting at that position. E.g., '8.1' scans as one number lexeme and does NOT include a '.' operator.

If no other lexeme can be found, the next character is a 1-character 'illegal lexeme'. Note that this produces some idiosyncratic results. For example, if you forget the closing " in a quoted string, there is no quoted string lexeme, and the " starting the string becomes a 1-character illegal lexeme. Similarly if you put an illegal character representative, such as \h, in a quoted string. To be sure you implement the above rules precisely, you should carefully check that your solution gets the Sample Output below when given the Sample Input below.

Input

For each of several test cases two lines. The first line is the test case name. The second line is the line you are to scan into lexemes.

There are NO tab characters in the input, so the only space characters in the input are single space characters and line ending line feeds. No line is longer than 80 characters (not counting line feeds).

Input ends with an end of file.

```
Output
------

For each test case, first an exact copy of the test case
name line.  Then two lines.  The first is a copy of the
input line to be scanned with '|' marks inserted at the
beginning and end and in between scanned lexemes.  The
next line has under each lexeme character a letter
giving the lexeme type.  This letter is simply the first
letter of the lexeme type name (i.e., 's' for symbol,
'o' for operator, 'i' for illegal lexeme, etc.).

Remember to test your program on the Sample Input and
be sure its output EXACTLY matches the Sample Output.
Note that numbers and symbols can be arbitrarily long,
and numbers CANNOT begin or end with '.'.  Also illegal
quoted strings are NOT recognized as quoted strings, and
their initial " is treated as an illegal lexeme.
Lastly, illegal lexemes are all 1-character lexemes,
like punctuation and operators.


Sample Input
------ -----

-- SAMPLE 1 --
x  =  5y21 + 3*x+foo("hi\\n",7.8);
-- SAMPLE 2 --
7.8 7. 8 7 .8 01234567890123456789.x!!
-- SAMPLE 3 --
?"He said: \"Ha\"?\\n" + "He He\n" + "Ho"
```

```
Sample Output
------ ------

-- SAMPLE 1 --
|x|  |=|  |5|y21| |+| |3|*|x|+|foo|(|"hi\\n"|,|7.8|)|;|
 s ww o ww n sss w o w n o s o sss p qqqqqqq p nnn p p
-- SAMPLE 2 --
|7.8| |7|.| |8| |7| |.|8| |01234567890123456789|.|x|!|!|
 nnn w n o w n w n w o n w nnnnnnnnnnnnnnnnnnn o s i i
-- SAMPLE 3 --
|?|"He said: \"Ha\"?\\n"| |+| |"|He| |He|\|n|" + "|Ho|"|
 i qqqqqqqqqqqqqqqqqqqq w o w i ss w ss i s qqqqq ss i



File:      lexemes.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Tue Oct 12 18:39:49 EDT 2010

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.
```

Snow Maze Luck
---- ---- ----

The scouts have a fun winter game called ‘Snow Maze’. They find a bunch of trees and put a target on each. Next to each targeted tree they place two arrows, one red and one green, each pointing at another targeted tree, though this second tree can often not be seen from the first tree, due to the other non-targeted trees in the woods. One of the trees is designated as the start, and another as the goal.

To progress through the maze a scout fires a snowball at the target on the tree the scout is currently at. If the scout hits the bulls eye, the scout goes next to the targeted tree pointed at by the green arrow beside the tree the scout is at. If the scout misses, the scout follows the red arrow. In each case, the scout must use a compass to go in a straight line through the forest until the scout finds the next targeted tree.

When setting up the contest, the organizers want to know the minimum distance a scout would travel if the scout hit H bulls eyes and missed the rest of the time and was maximally lucky as to which targets the scout hit.

The first step is computing the distances between each targeted tree and its red and green arrow next targeted trees. The second step is computing the minimum distances for each H. The first step has been done for you: you must do the second step.

Input

For each test case:

Line 1: The test case name.

Line 2: The number N of targeted trees. The trees are numbered 1, 2, ..., N. Tree 1 is always the start tree, and tree N the goal. 2 <= N <= 50.

Lines 3..N+1:
One line for each tree T in the order from tree T=1 to tree T=N-1. Tree T’s line contains the red target next tree number R, the distance from T to R, the green target next tree number G, and the distance from T to G, in order. All numbers are integers. Distances are from 1 through 10. Note there is no line for T=N.

The input ends with and end of file.

In the data that will be use to test your solution, the distances between trees are not required to make geometric sense: e.g., tree 1 can be 1 unit from tree 2 and tree 2 can be 2 units from tree 3 while tree 1 is 8 units from tree 3. Also, the red and green arrows of a tree can both point at the same tree.

```
Output
------

Two lines for each test case.  The first line is an
exact copy of the test case name line.  The second line
contains N+1 values that correspond to H=0, H=1, ...,
H=N from left to right.  For each H, the corresponding
value is the the minimum distance of a path with H bulls
eyes; or is 'X' if no such path exists.

Note that a path can loop back on itself.  However,
once a path arrives at tree N, the path must stop.


Sample Input
------ -----

-- SAMPLE 1 --
4
2  1 4 8
1 10 3 1
4 10 4 10
-- SAMPLE 2 --
5
3 1 2 1
4 2 1 1
5 4 2 6
1 1 3 2


Sample Output
------ ------

-- SAMPLE 1 --
X 8 12 X X
-- SAMPLE 2 --
5 9 7 11 9 13
```

File: snowluck.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Sun Mar 20 09:32:49 EDT 2016