Problems Index Wed Oct 10 01:03:30 PM EDT 2007

BOSPRE 2007 Problems.

The problems are in approximate order of difficulty, easiest first.

```
    problems/piglatin
        High school keeps coming back.

    problems/antique
        Did the Romans have it right?

    problems/shootemup
        A little intelligence goes a long way.

    problems/tx0r
        Antique computers fascinate some people.

    problems/bending
        Sea legs for the land decks.

    problems/cliques
        Networking has its clusters.
```

Pig Latin

You have been asked to translate English words to Pig Latin. The translation is very simple: take all the consonants at the beginning of the word, move them to the end, and add ‘ay’. If there are no consonants at the beginning of the word, just add ‘ay’ to the end. The consonants are all letters except ‘a’, ‘e’, ‘i’, ‘o’, ‘u’, and ‘y’. Note that ‘y’ is NOT a consonant for our purposes.

Input

A sequence of lines each containing an English word. There are no spaces in any line. Words will contain only lower case letters.

The input ends with an end of file.

Output

For each English word, one line containing nothing but the translation of the word into Pig Latin.

Example Input
------- -----

```
you
help
me
to
understand
pig
latin
this
hour
```

Example Output
------- ------

```
youay
elphay
emay
otay
understanday
igpay
atinlay
isthay
ourhay
```

Note: Actual Pig Latin moves only initial consonant SOUNDS, and therefore does not move unsounded initial consonants. Thus ‘hour’ would become ‘houray’ in actual Pig Latin. There are also variants which put ‘way’ or ‘yay’ or some such at the end of words that begin with a vowel sound.

```
File:       piglatin.txt
Author:     Bob Walton <walton@deas.harvard.edu>
Date:       Wed Oct 10 03:31:33 EDT 2007

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

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2007/10/10 07:33:35 $
    $RCSfile: piglatin.txt,v $
    $Revision: 1.3 $
```

```
Antique Formatting
------------------

You have been asked to write prototype a formatting
function.  As a test your program is to read a format-
ting string and some arguments, and apply the formatting
string to the arguments.

You boss, however, is a bit on the antique side.

The formatting string is a sequence of commands, each
of which is a single character.  For the prototype,
only two commands are implemented:

    w   Print the next argument as a string of words.

    i   Print the next argument as an integer, treating
        the formatted integer as one word.

However, as we said, your boss is antique.  Words are to
be printed from right to left in the output.  An integer
is to be printed as a Roman numeral word.  For this
prototype, each formatting string produces one line of
output, with words separated by single spaces, and no
other spaces in the line.

More specifically, the words in a ‘w’ string are separa-
ted by single spaces, but these spaces are discarded
after extracting the words.  The order of the words in
the string is the reverse of the order of the words in
the output line.  However, the letters within a printed
word are in the same order as the letters within the
word in the string.  So word order is reversed but
letter order is not.

Roman numerals use the following letters to represent
numbers:

        I       1
        V       5
        X       10
        L       50
        C       100
        D       500
        M       1000

There is no way to represent zero, and you will not be
asked to print zero.

The first 10 numbers are encoded as:

        I       1
        II      2
        III     3
        IV      4       (-1 + 5)
        V       5
        VI      6
        VII     7
        VIII    8
        IX      9       (-1 + 10)
        X       10

Your boss wants you to simply encode the digits of the
integer using the encodings just given.  For the tens
digit you simply make the replacements:

        I --> X
        V --> L
        X --> C
```

```
and for the hundreds digit

        I --> C
        V --> D
        X --> M

You will not be asked to print any number larger than
3999, which allows you to use M, MM, or MMM for the
thousands digit.

Note the order that digits are printed in is the same
for decimal and our Roman numerals.  The thousands digit
is printed before the hundreds digit, etc.

For example, 1999 is printed as MCMXCIX, and 1849 as
MDCCCXLIX.

Input
-----

For each test case, one line containing the name of the
test case, followed by one line containing the format-
ting command string, followed by one line for each
argument containing just the argument, followed by an
empty line.  Input integers are represented in decimal,
and are in the range from 1 to 3,999.

Input ends with an end of file.

Output
------

For each test case, one line containing the name of the
test case, followed by the output line for that test
case.

No output line will be longer than 80 characters.
```

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

TEST-1
wiw
we want
4
words

TEST-2
iwiwi
2
plus
2
equals
4

TEST-3
wi
a good year is
1999

TEST-4
wi
another year is
1849

TEST-5
wi
the last year of the millennium is
2000


[Note the last line of the input is empty.]
```

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

TEST-1
words IV want we
TEST-2
IV equals II plus II
TEST-3
MCMXCIX is year good a
TEST-4
MDCCCXLIX is year another
TEST-5
MM is millennium the of year last the
```

File: antique.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Wed Oct 10 12:05:13 EDT 2007

RCS Info (may not be true date or author):

```
    $Author: walton $
    $Date: 2007/10/10 16:05:57 $
    $RCSfile: antique.txt,v $
    $Revision: 1.5 $
```

The Land of Shot-Em-Up

In the Land of Shot-Em-Up conflicts are settled by robot duels. Each side builds a robot. The two robots are turned loose, on their own, in an arena, and whichever robot kills (makes non-functional) the other robot wins for its side of the conflict.

You have entered 'Conflict School' in the land of Shot-Em-Up, and are taking a (very) beginning course in programming these robots. Your first assignment is to program a simple robot that will do battle in a virtual world.

The virtual world is a 10x10 board of squares. The two robots take turns making moves. A move is one of:

    moving one square in any of the 8 directions
    (4 of which are diagonal)

    staying put at the robot's current location

    shooting in any of the 8 directions,
    while staying put at the current location

If you move off the edge of the board you die.

If you shoot in a direction the shot hits anything in its path at the time of the shot. A robot can absorb 2 hits without ceasing to function, but will die when it receives a 3'rd hit. However, if you shot in your last move, your shot in this move will have too little power to hit anything. Your 'gun' takes one move without shooting to 'recharge'.

Somewhat oddly both robots can occupy the same square of the board at the same time. If one of the robots shoots at such a time, the shot never hits the other robot.

The robots cannot see each other. They can tell, however, when they are hit, and from what direction the shot that hit them came.

Your assignment is to write a robot program good enough to beat a given opponent at least 51 out of 100 times. The opponent is a stupid random robot provided by the teacher.

Your program is not run directly, but instead is run by another program called 'arena'.

Your Program's Input

The 'arena' program writes lines that appear in your program's standard input. These lines, which give you information, have the following format:

    P x y

        Prepare to start a new combat. You are on board
        square (x,y). Here x and y are integers, with
        0 <= x,y <= 9.

    H dx dy

        Your opponent shot you in his last move. The
        shot passed through square (x+dx,y+dy) on its
        way to hitting you, where you are currently at
        square (x,y), dx and dy are integers, dx and
        dy are not both 0, and -1 <= dx,dy <= 1.

    N

        Your opponent did not shot you in his last move.

W

Your opponent died in your or his last move, and you have won. The combat is over.

L

You died in your or your opponents last move, and you have lost. The combat is over.

D<anything>

This line is a debugging instruction for your program. You produce such lines by giving them as input to the arena program: see below.

After inputting a P, N, or H line, you must make a move by outputting an M or S line, as specified below. After inputting a W or L line, you should read another line. The next thing input will be a P line or the end of file. After inputting a D line, you should do what the D line tells you to (you decide what this is), and then input another line. You can output / lines anytime (see below) with debugging information.

There are no superfluous space characters on any input line. The board squares are numbered (0,0) at the upper left to (9,9) at the lower right. To make a move, you output an M or S line (see below), and then you read a line of input to find out what happened next.

Your program should terminate when it reads an end of file.

Your Program’s Output

Your program writes lines to its standard output that are read by the ‘arena’ program. These lines announce your moves, and have the following formats:

M dx dy

Move from your current board square (x,y) to the board square (x+dx,y+dy), where dx and dy are integers and -1 <= dx,dy <= +1. dx = dy = 0 is permitted, and is used to implement the ‘staying put’ move. 0 <= x+dx,y+dy <= 9 is required (else you die and lose).

S dx dy

Shoot. The shot starts at your current square (x,y) and goes in a straight line through the square (x+dx,y+dy) and on to the edge of the board. Here -1 <= dx,dy <= +1, and dx,dy are both integers. dx and dy may NOT both be 0. You are allowed to shoot off the edge of the board, e.g., x+dx > 9 is allowed, but you will not hit anything.

/<anything>

This is a comment line. It is output by the arena program, and may be used for debugging. E.g., you may output / lines in response to a D line.

You cannot move and shoot at the same time.

Arena Input

The ‘arena’ program reads commands from its standard input, which is normally the shootemup.in file. These commands define test cases and debugging options.

G n — Reset the random number generator seed to n, which must be an unsigned integer with at most 9 digits. The random number generator is used by your opponent, and is used to determine your initial position. If you want your opponent to behave differently, or to run rounds differently, you must input a different seed.

-<anything> — Start a new combat. This line is echoed to the standard output and serves to name the combat. This is the first input line describing a combat, excepting those combats conducted by an R command.

\+ — Make a pair of moves, one for you and one for your opponent.

. — Continue the combat to the end.

B — Display the board. Good for debugging. On the board, ‘Y’ is you after your last move, ‘O’ is your opponent at the same time, ‘+’s mark your shot if you shot in your last move, ‘-’s mark your opponent’s shot if it shot in its last move.

B1 — Turn on display of the board after every move of your opponent.

B0 — Turn off ditto.

*<anything> — Comment line. Echoed to standard output.

D<anything> — This line is sent to your program. It can be used to trigger a debugging action: see above.

R n — Run a round of n combats, and print a round line at the end. The round line has the form

```
! ROUNDS r WINS w LOSES l ERRORS e j
```

where

- r is the number of rounds
- w is the number of rounds that ended in wins for you
- l is the number of rounds that ended in loses for you
- e is the number of rounds that ended when your program made and error (explained in ‘*’ lines)
- j is the judgment, which is
  - ‘PASS’ if 2w > r
  - ‘FAIL’ if 2w <= r

Unless a combat initiated by a ‘-’ command is in progress, only G, R, and ‘-’ commands are executed.

Example Arena Input
------- ----- -----

The following can be put in the shootemup.in file.

```
G 55
-TEST 1
B1
+
+
+
.
R 100
R 100
R 100
```

Arena Output

The 'arena' program writes output to its standard output, which is normally put in the shootemup.out file.

The 'arena' program echos all its input lines, all the lines it sends to your program, and all the lines it receives from your program. As an exception, the lines your program sends arena and that arena sends your program are NOT echoed during the R command.

The arena program outputs lines beginning with '!' that contain error messages, board positions, etc.

If you execute

```
        arena shootemup < xx.in > xx.out
        arena shootemup < xx.out > foo
```

Then foo and xx.out should be identical. That is, the the second command will repeat the moves made by the first command.

Similarly, to replay a game in the debugger you execute

```
        grep '^[PHNWLD]' xx.out > xx.din
        gdb shootemup
        run < xx.din
```

```
Example Arena Output
------- ----- ------

G 55
-TEST 1
P 5 6
B1
+
S 0 1
N
!
! ...-......
! ....-.....
! .....-....
! ......-...
! .......-..
! ........O.
! .....Y....
! .....+....
! .....+....
! .....+....
!
[[ Substantial output omitted here ]]
R 100
! ROUNDS 100 WINS 91 LOSES 9 ERRORS 0 PASS
R 100
! ROUNDS 100 WINS 87 LOSES 13 ERRORS 0 PASS
R 100
! ROUNDS 100 WINS 92 LOSES 8 ERRORS 0 PASS
```


File: shootemup.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Wed Oct 10 12:44:10 EDT 2007

RCS Info (may not be true date or author):

```
    $Author: walton $
    $Date: 2007/10/10 16:44:21 $
    $RCSfile: shootemup.txt,v $
    $Revision: 1.9 $
```

```
The TX-0 Reincarnate
--------------------

The TX-0 computer was built in 1955 as an experimental
computer to test transistor circuitry, which was new to
computers at that time.  Its instructions contained a
2-bit operation code and a 16-bit address.  The word
length was 18-bits.  The computer had a 1-word
accumulator and up to 65536 words of random access
magnetic core memory.

The TX0R computer is very similar, but has been adapted
for use in programming contests.  Its instruction set
is:

    STORE    address    Store accumulator in the word at
                        the given address.

    ADD      address    Add the word at the given
                        address to the accumulator.

    TRANSFER address    Go to the instruction at the
                        given address if the accumulator
                        is negative.

    OPERATE  source,operation,destination

                        Input the source word, perform
                        the indicated operation on it,
                        and output the result to the
                        destination.

        sources:

            AC          Accumulator

            READ        Read the next row of the input
                        tape, and interpret that row as
                        a word value.  Or use the value
                        0 if the tape is at its end.

            EOF         The value 0 if the last READ
                        command read the next row, and
                        the value -1 if it did not
                        because the tape was at its end.

        operations:

            COPY        Copy the word

            CLEAR       Zero the word

            NEGATE      Negate the word

        destinations:

            AC          Accumulator

            HALT        Halt normally and display result

            ERROR       Halt indicating error; the
                        result value is ignored

The memory of this computer consists of 256 32-bit
words.  Both program and data must be stored in this
limited memory.

The words are formated as 2’s complement integers (just
like 32-bit words in a modern computer).  Each
instruction takes one word (most of which is unused).
You are to use an assembler and therefore do not need to
know the precise instruction format.
```

The input is a punched paper tape with 32 columns. Every time READ is used as a source to an OPERATE instruction, the next row of the tape is read, thereby reading a 32 bit word. If there is no next row (because the paper tape reader is at the end of tape), 0 is read. The EOF source to the OPERATE command produces the value 0 if a row was read by the last READ source to an OPERATE command, and produces -1 if no row was read because the tape was at its end.

The original TX-0 tape was had just 6 columns and reliability concerns, but the TX0R paper tape has 32 columns and is completely reliable.

Programs

A TX0R program is written in a file whose name has the .tx0r extension. A program is assembled, and consists of a sequence of word descriptions, each on one line. The possible word description lines are:

```
    [label:]    STORE    address
    [label:]    ADD      address
    [label:]    TRANSFER address
    [label:]    OPERATE  source,operation,destination
    [label:]    WORD     value
```

STORE, ADD, TRANSFER, and OPERATE are instructions. You need not know the format of instruction words, as you will be using an assembler.

WORD describes a word whose initial value, at the beginning of program execution, is given.

Addresses and values may be integers or symbolic names, where a symbolic name is a sequences of letters, digits, and underbars, beginning with a letter or underbar. The first word description is for the word at location 0, the second for the word at location 1, etc. Any label given is a symbolic name that denotes the location of the word described on the same line. A label may be used as an address or value.

For the OPERATE class instruction, sources, operations, and destinations are named as indicated above. See the example below.

Blank lines in the input are ignored. The characters ‘//’ and anything following them in a line are ignored; so comments begin with ‘//’.

Program execution begins at word 0.

Example Program
------- -------

This program reads the data input and outputs the sum of all the input values.

```
           OPERATE      AC,CLEAR,AC     // sum = 0
           STORE        sum
loop:      OPERATE      READ,COPY,AC    // datum = READ
           STORE        datum
           OPERATE      EOF,COPY,AC     // if EOF goto
           TRANSFER     end_loop        //     end_loop
           OPERATE      AC,CLEAR,AC     // sum += datum
           ADD          datum
           ADD          sum
           STORE        sum
           OPERATE      AC,CLEAR,AC     // goto loop
           ADD          minus_one
           TRANSFER     loop
end_loop:  OPERATE      AC,CLEAR,AC     // HALT sum
           ADD          sum
           OPERATE      AC,COPY,HALT
sum:       WORD         0
datum:     WORD         0
minus_one: WORD         -1
```

Input

You will be using a simulator that executes a program under the direction of an input file.

The input file consists of any number of test cases. Each test case begins with a line that contains nothing but the test case name. This name must begin with a letter.

After the test case name are the contents of the input data tape that the program can read using the READ source to the OPERATE instruction. These contents consist of a sequence of zero or more integers.

Each test case ends just before the end of file or the next line beginning with a letter.

The input ends with an end of file.

Output

For each test case the simulator outputs one line containing the test case name, as input, and one line containing one of the following:

```
        HALT result
        ERROR
```

The result is an integer, printed with no spaces or high order zeros. There is a single space character before this result, and no other space characters in the line. The words HALT and ERROR are the destination of the OPERATE instruction that halted the program.

```
Simulation Command
---------- -------

You can use the ‘make’ and ‘make debug’ commands to
run your program, or you can run your program directly
with the command:

    tx0r_simulator [-debug] tx0r.tx0r

This assembles the TX0R code in the file tx0r.tx0r and
runs it according to the instructions in the standard
input, which contains the input file.  -debug prints an
assembly listing, and after each instruction execution,
prints the pc and ac.  The simulator writes results to
the standard output.


Problem
-------

You are to write a program in the TX0R language in the
file tx0r.tx0r.  This program reads the input data tape
and HALTs displaying the difference between the smallest
datum and the largest datum.  It is an error if there
are no data values (empty input tape), and in this case
the program should execute an OPERATE instruction with
ERROR destination.
```

```
Example Input
------- -----

TEST 1
1
2
3
4
5
TEST 2
TEST 3
5
-6
7
8
2
-3


Example Output
------- ------

TEST 1
HALT 4
TEST 2
ERROR
TEST 3
HALT 14
```

```
File:       tx0r.txt
Author:     Bob Walton <walton@deas.harvard.edu>
Date:       Wed Oct 10 07:30:22 EDT 2007

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

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2007/10/10 11:36:47 $
    $RCSfile: tx0r.txt,v $
    $Revision: 1.5 $
```

Bending Deck Boards

Robert is building a deck on the side of his house and has a problem. He is using ‘composite boards’ for the floor of the deck, but these, being made of the plastic polyethylene and wood fiber, expand much more than wood when the temperature gets hot. Robert is afraid the boards will bend, or warp, because of this expansion. So he does a simple calculation to see how bad the problem is. You are being asked to program this calculation.

A section of board is normally a straight line between two points, B and C, at which the board is fastened by screws to joists. Suppose we have such a section of length L, that is, the distance from B to C is L. Suppose the section length changes by expansion to L+y, where y > 0 is a small number, but the end points of the section remain anchored at B and C. The section must assume a non-straight-line shape. Assume it becomes an arc of a perfect circle, with end points B and C. Let the circle have radius R. Note that R is determined by L and y.

The straight line from B to C is then the chord of the circle from B to C. Let x be the maximum distance between a point on the arc and a point on the chord. y measures the amount of expansion, and x measures the amount of bending.

Note that if you change the scale of the situation by multiplying all distances by a constant C, the circle remains a circle but now of radius C*R, the chord remains a chord but now of length C*L, the arc remains an arc but now of length C*(L+y), and the maximum distance between a point on the chord and a point on the arc is now C*x. Therefore, x/L as a function of y/L does not depend on L. So you are asked to find this function.

Note that given R you can compute x and y. Also, R decreases whenever y increases, and y/L as a function of R/L does not depend on L. The problem reduces to computing R/L from y/L by inverting a monotonic function.

Input

For each of several cases, a line containing a value for y/L. No lines contain any spaces. The input terminates with an end of file.

Output

For each case a line containing in order:

```
        the value of y/L
        a single space
        the value of R/L
        a single space
        the value of x/L
```

Print all values with exactly 8 decimal places. Do not include any extra spaces. Use double precision floating point arithmetic for all computations.

```
Example Input
------- -----

0.01
0.02


Example Output
------- ------

0.01000000 2.06885226 0.06132899
0.02000000 1.48249945 0.08686174


File:      bending.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Wed Oct 10 07:12:26 EDT 2007

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

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2007/10/10 11:16:42 $
    $RCSfile: bending.txt,v $
    $Revision: 1.3 $
```

```
Cliques
-------

Given an undirected graph G (a set of vertices and
edges), a clique C in G is a set of vertices each pair
of which is joined by an edge of G.

Given a graph G, you are asked to find all the cliques
of G with at least 3 vertices.

We restrict ourselves to graphs with at most 26 nodes
that are labeled A through Z.  We represent edges by
words consisting of two letters, where the letters are
in alphabetical order.  For example, AX represents an
edge, but XA does not.  We represent cliques by words
that list all the vertices in a clique in alphabetical
order.  For example, AXZ might represent a clique in
some graph, but AZX could NOT be a legal clique
representative.

To represent a graph or a set of cliques, we list repre-
sentatives of edges of the graph, or of the cliques in
the set, lexicographically: that is, in dictionary
order.  See the examples below.


Input
-----

For each of several cases, a specification of a
graph G as follows:

   A line containing the name of the graph.

   A line containing the number n of edges.

   n lines each containing nothing but a two letter word
     representing an edge.  No edge will be repeated,
     and the edge representatives will be sorted
     lexicographically.

Only the graph name line may contain any spaces.  The
input terminates with an end of file.


Output
------

For each case, a single line containing the name of the
graph exactly as input, followed by one line for each
clique with 3 or more vertices.  The line for a clique
contains just the representative of the clique, and all
the clique lines for one graph are sorted lexicographic-
ally.


Example Input
------- -----

TEST 1
6
AB
AC
AD
BC
BD
CD
TEST 2
10
AE
AF
BE
BF
BX
EF
EX
FX
XY
YZ
```

```
Example Output
------- ------

TEST 1
ABC
ABCD
ABD
ACD
BCD
TEST 2
AEF
BEF
BEFX
BEX
BFX
EFX
```

File: cliques.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Wed Oct 10 07:18:28 EDT 2007

RCS Info (may not be true date or author):

```
    $Author: walton $
    $Date: 2007/10/10 11:28:01 $
    $RCSfile: cliques.txt,v $
    $Revision: 1.3 $
```