Problems Index Wed Oct 10 04:55:48 EDT 2012

BOSPRE 2012 PROBLEMS
------ ---- --------

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

```
    problems/spoonerisms
        A switch causes a stitch.
        Boston Preliminary 2012

    problems/fractals
        Increasing dimensions only slightly.
        Boston Preliminary 2012

    problems/pagerank
        Whats popular these days?
        Boston Preliminary 2012

    problems/lambdadev
        Safe reduction.
        Boston Preliminary 2012

    problems/gorillatoe
        Darn gorilla stepped on my toe!
        Boston Preliminary 2012

    problems/anglepuzzle
        A protractor mess.
        Boston Preliminary 2012

    problems/broppers
        Sci Fi for whiz kids.
        Boston Preliminary 2012
```

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

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

    $Author: walton $
    $Date: 2012/10/01 09:19:56 $
    $RCSfile: spoonerisms.txt,v $
    $Revision: 1.8 $

Fractals

One way of making fractals is to take a line drawing and repeatedly perform a self-similar replacement operation on the line segments. A self-similar operation is one that is invariant under scaling, rotation, and translation (and sometimes reflection).

To be specific, the line segment replacement operation is defined by giving the line segments that will replace a unit line segment. The sequence of line segments that replace the unit line segment is called a ‘generator’. Any line segment can be made by scaling, rotating, and translating the unit line segment, so the replacement of any line segment L can be calculated by scaling, rotating, and translating the generator in the same way that the unit line segment was scaled, rotated, and translated to make L. When scaling, all dimensions must be scaled equally, and reflections are not allowed.

The line segments are directed; each has a beginning and an end, and these CANNOT be switched.

To be even more specific, suppose the unit line segment

```
                (0,0) - (1,0)
```

is replaced by the generator

```
    (0,0) - (1/3,0) - (1/2,sqrt(3)/6) - (2/3,0) - (1,0)
```

This is the ‘Koch generator’, and consists of dividing the unit line segment into thirds, constructing an equilateral triangle with the middle third as a side, and erasing the middle third of the original line.

To apply the Koch generator to the line segment

```
        L = (5,-2) - (7,0)
```

we first make L from the unit segment by scaling the unit line segment by sqrt(8), then rotating counter-clockwise by 45 degrees, and lastly translating by (5,-2). Then we do the same to the Koch generator, and get 4 replacement line segments for L. See the first sample below.

Fractals are made by applying generator defined replacements to all the line segments in a line drawing, and then repeating this entire operation an infinite number of times. You have been asked to do this, but just a finite number of times.

The Koch generator is oriented. This means that if a line drawing segment has its beginning and ending switched, its Koch generator generated replacement will be a reflection of the original replacement. Specifically, the triangle will move to the other side of the original line. An oriented generator cannot sensibly be applied to a line drawing whose lines are not orientable. Some generators a not oriented, and can be applied to any line drawing.

Note also that the Koch generator is a connected curve from (0,0) to (1,0), but this is not required; a generator can be any set of line segments, possibly disjoint and possibly intersecting.

Input

For each of several test cases, first a line containing the test case name. Then one or more lines of the format

```
        x1 y1 x2 y2
```

each describing one line segment (x1,y1) - (x2,y2) of the generator that replaces (0,0) - (1,0). Then a line containing just ‘*’ to signal the end of the generator. Next, more lines of the above format describing the line segments of the line drawing, followed by another line containing just ‘*’. The test case ends with a line containing just a single integer N, specifying the number of iterations of the replacement operation.

The generator will contain between 1 and 100 line segments, the line drawing will contain between 1 and 100 line segments, N will be between 0 and 10, and no line will longer than 80 characters.

Input ends with an end of file.

Output

For each test case, first a copy of the test case name line, then lines describing the line segments resulting from N replacements, and then a line containing just ‘*’. Each line that describes a line segment has the same format as in the input, and the numbers output in the line must be accurate to least 3 decimal places.

Initially the line drawing segments input are the current line segments, and these are in an ordered sequence. One iteration replaces EACH current line segment in order by the generator defined replacement. ORDER MUST BE MAINTAINED. There are N iterations. Note that N == 0 is possible (usable to display the input: see next paragraph).

The output may be printed as a graph or displayed in an X-window by the commands:

```
        print_fractals
        display_fractals
```

provided the output of your program has been stored in the file ‘fractals.out’. To see the sample output instead use the commands

```
        print_fractals sample.test
        display_fractals sample.test
```

(here sample.test is the output for sample.in).

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

-- KOCH CURVE; scale sqrt(8); rotate 45 deg --
0 0 0.3333333333 0
0.3333333333 0 0.5 0.28867513
0.5 0.28867513 0.6666666667 0
0.6666666667 0 1 0
*
5.00000000 -2.00 7 0
*
1
-- KOCH CURVE; 4 iterations --
0 0 0.3333333333 0
0.3333333333 0 0.5 0.28867513
0.5 0.28867513 0.6666666667 0
0.6666666667 0 1 0
*
0 0 1 0
*
4
-- KOCH FLAKE; 0 iterations --
[See sample.in file for rest of input]
```

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

-- KOCH CURVE; scale sqrt(8); rotate 45 deg --
5.000 -2.000 5.667 -1.333
5.667 -1.333 5.423 -0.423
5.423 -0.423 6.333 -0.667
6.333 -0.667 7.000 0.000
*
-- KOCH CURVE; 4 iterations --
0.000 0.000 0.012 0.000
0.012 0.000 0.019 0.011
0.019 0.011 0.025 0.000
0.025 0.000 0.037 0.000
0.037 0.000 0.043 0.011
0.043 0.011 0.037 0.021
0.037 0.021 0.049 0.021
[see sample.test file for rest of output]
```

Reference

See Chapter 1 of ‘Fractals, Chaos, and Power Laws’ by Manfred Schroeder.

The formal definition of a ‘fractal’ is ‘a set whose fractal dimension exceeds its topological dimension’. However, the term ‘fractal dimension’ refers to one of many not exactly equivalent ways of computing dimension. If we use generators that are connected curves and apply them an infinite number of times we can generate a fractal whose topological dimension is 1. If we use Hausdorff dimension and the Koch generator the fractal dimension is log(4)/log(3). There are many other ways of generating fractals of topological dimension 1.

File: fractals.txt
Author: Bob Walton <walton@seas.harvard.edu>
Date: Wed Oct 10 04:20:40 EDT 2012

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

```
    $Author: walton $
    $Date: 2012/10/10 08:22:04 $
    $RCSfile: fractals.txt,v $
    $Revision: 1.10 $
```

PageRank

Google uses a page ranking algorithm to determine the importance of each web page. The rank of a page is the probability that a particular random web surfer will be looking at the page at any given moment.

Suppose we represent the web as a graph with N nodes, each representing a page, and with a directed edge representing each link from one page to another.

The random surfer in question behaves as follows.

The surfer chooses a first page randomly from among the N pages.

To move from a current page to the next page, the surfer does the following. If the current page is the source of zero links, the surfer chooses the next page randomly from among the N pages of the web. Otherwise, with probability 1-ALPHA, the surfer does the same thing (as if the page sourced zero links), and with probability ALPHA, the surfer chooses a link sourced at the current page at random and follows that link.

When choosing from among a set of pages or links at random the surfer gives equal probability to each page or link that might be chosen. So to choose at random from among the N pages of the web, each page has probability 1/N of being chosen. And to choose at random from among Q links sourced at the current page, each link has probability 1/Q of being chosen.

You have been asked to compute the probability for each page that that page will be the K'th page visited by the surfer, for a given web graph and value of K.

Input

For each of several test cases, first a line containing the name of the test case, and then a line containing

```
N ALPHA K
```

where N is the number of pages and K is the number of pages the surfer is to visit. After these two lines are N lines each containing a list of page numbers followed by a 0. Pages are numbered 1, 2, ..., N. The i'th of these lines contains the numbers of the pages targeted by links sourced at page i.

So in the sample input below, node 1 has two links to node 2, node 2 has links to nodes 1, 2, and 3, and node 3 has zero links. Note that a node may have several links to the same target node and may have links to itself.

1 <= N <= 100; 0 <= ALPHA <= 1; 1 <= K <= 10,000.

Input ends with an end of file.

Output

For each test case, one line containing the name of the test case (copied exactly from the input), followed by N lines each containing a page number i followed by the probability the K'th page surfed will be page i. The page numbers must be in increasing order. The probabilities must be output with exactly 6 decimal places.

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

-- SAMPLE 1 --
3 0.00 2
2 2 0
3 2 1 0
0
-- SAMPLE 2 --
3 1.00 2
2 2 0
3 2 1 0
0
-- SAMPLE 3 --
3 0.85 3
2 2 0
3 2 1 0
0

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

-- SAMPLE 1 --
1 0.333333
2 0.333333
3 0.333333
-- SAMPLE 2 --
1 0.222222
2 0.555556
3 0.222222
-- SAMPLE 3 --
1 0.265648
2 0.468704
3 0.265648
```


```
Reference:
    "PageRank: Standing on the Shoulders of Giants",
    by Massimo Franceschet, Communications of the ACM,
    Vol 54, No 6, June 2011, pp 92-101.
```



```
File:       pagerank.txt
Author:     Bob Walton <walton@seas.harvard.edu>
Date:       Sat Oct  6 03:26:45 EDT 2012
```



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

    $Author: walton $
    $Date: 2012/10/06 07:28:27 $
    $RCSfile: pagerank.txt,v $
    $Revision: 1.5 $
```

```
Lambda Developments
------ ------------

The beta reduction rule in the lambda calculus can lead
to infinite reductions, as in the case

    (\x.xx)(\x.xx) => (\x.xx)(\x.xx)

where we use ‘\’ in place of the Greek letter lambda.

Recall that a lambda term of the form \x.M is called an
‘abstraction’ and one of the form (\x.M)N is called a
‘redux’ because it can be beta reduced to M[x:=N] which
denotes the result of substituting N for x in M.  Also
a term of the form MN is called an ‘application’.

But if we mark the reduxes in a lambda term and only
reduce the marked reduxes and copies of the marked
reduxes made by previous reduction steps, then infinite
reductions are no longer possible.

You have been asked to check this out in some important
special cases.

We will mark a redux with an integer put just after the
‘\’ in the abstraction term of the redux.  Thus we have

    (\1x.xx)(\x.xx) => (\x.xx)(\x.xx)

but after this one reduction there are no marked reduxes
left, because the redux on the right is ‘created’ when
the argument of the redux on the left, (\x.xx), is
substituted for the first x in the xx in the abstraction
of the redux on the left, and created reduxes are not
marked.  Created reduxes are never marked, but copied
ones are, as in

    (\1x.xx)((\2x.x)(\y.y)) =>
    ((\2x.x)(\y.y))((\2x.x)(\y.y)) =>
    ((\y.y)((\2x.x)(\y.y)) =>
    ((\y.y)(\y.y))

Note that only reduxes and not abstractions are being
marked, even though the mark is put in the abstraction
part of the redux.  Thus (\1x.xx)(\2x.xx) is NOT syntac-
tically legal because (\2x.xx) is not the ABSTRACTION
TERM IN A REDUX (it is the argument term in a redux).

Suppose we start with an initial unreduced term that
obeys the following:

(A) The marked reduxes of the term have distinct marks.

(B) Every abstraction in the term (marked or unmarked)
    has a different bound variable name.

(C) All free variable names in the term are distinct
    from all bound variable names in the term.

Then it can be proved that if we reduce only marked
reduxes then:

(1) At any stage all reduxes with the same mark are
    disjoint.

(2) In any marked redux reduction ((\nV.M)N)===>M[V=N]
    the free variables of N are distinct from the bound
    variables of (\nV.M), so bound variable names do NOT
    need to be changed in order to avoid ‘variable
    capture’.

(3) The reduction is terminating, meaning that reduction
    of only marked reduxes cannot continue indefinitely.
```

Let =>N denote the beta reduction of all reduxes with mark N (see Sample Output). You are given a marked lambda term obeying (A), (B), (C) above and are being asked to compute the results of using first =>1 to eliminate \1 reduxes, then =>2 to eliminate \2 reduxes, etc., until all marked reduxes are eliminated. Thus you can empirically test the theorem.

Input

For each of several test cases, first a test case name line, and then one line containing just a marked lambda term obeying (A), (B), (C).

The formal syntax of the marked lambda terms is

```
    T ::= V | (TT) | (\V.T) | ((\NV.T)T)     [terms]
    V ::= a | b | c | ... | x | y | z        [variables]
    N ::= 1 | 2 | 3 | ... | 7 | 8 | 9        [marks]
```

Note that abstractions and applications are surrounded by parentheses (unlike in the examples above).

There are no spaces in any input line other than the test case name lines. Lines are at most 80 characters long.

Input ends with an end of file.

Output

For each test case, first an exact copy of the test case name line, and then K+1 lines where K is the number of different marks in the case input term. The first line contains the input term followed by ' =>N' where N is the first marker in the input term, and markers are ordered numerically (1 before 2, etc.). The second line contains the term that results after doing all =>N reductions in the input term. If there is a second marker P in the input term, this is followed by ' =>P' and on the next line the result of doing all the =>P reductions after all the =>N reductions have been done. And so forth until all marked reductions have been done. The last line just contains the final unmarked term by itself (NOT followed by =>...).

IMPORTANT: You need not and MUST NOT make arbitrary substitutions for bound variables when computing reductions. As a result, the terms in the output are unique, and your output must match exactly the unique correct answer. Also, you must use the same syntax as in the input, and therefore include parentheses surrounding applications and abstractions. Lastly, you must not include any whitespace in your output lines (other than the test case name lines).

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

-- SAMPLE 1 --
((\x.x)y)
-- SAMPLE 2 --
((\1x.x)y)
-- SAMPLE 3 --
((\6x.(x(xy)))((\3z.(zw))w))
-- SAMPLE 4 --
((\2x.(x(xy)))((\3z.(zw))w))


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

-- SAMPLE 1 --
((\x.x)y)
-- SAMPLE 2 --
((\1x.x)y)=>1
y
-- SAMPLE 3 --
((\6x.(x(xy)))((\3z.(zw))w))=>3
((\6x.(x(xy)))(ww))=>6
((ww)((ww)y))
-- SAMPLE 4 --
((\2x.(x(xy)))((\3z.(zw))w))=>2
(((\3z.(zw))w)(((\3z.(zw))w)y))=>3
((ww)((ww)y))
```


File: lambdadev.txt
Author: Bob Walton <walton@seas.harvard.edu>
Date: Sat Oct 6 03:52:01 EDT 2012

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

    $Author: walton $
    $Date: 2012/10/06 10:08:55 $
    $RCSfile: lambdadev.txt,v $
    $Revision: 1.9 $

Gorilla Toe

Gorilla Toe is a game just like Tic Tac Toe with the following differences. First, the game is played on a board with N rows and M columns of squares, where 3 <= N,M <= 6. Second, to win you must occupy min(N,M) adjacent squares in a row or a column or a diagonal (so on a board with 3 rows and 5 columns you must occupy 3 adjacent squares in a row, a column, or a diagonal). Third, just before the game beings, a bunch of gorillas rushes out and occupies some squares, where they sit quietly during the rest of the game. You cannot occupy a square occupied by a gorilla - you just will not fit!

You have been asked to write a player's assistant which, when given a board state with P being the next player to move, will label each unoccupied square according to the fate of P if P moves next to that square and then both players play optimally.

As in Tic Tac Toe, the players are X who moves first and O who moves second.

Input

For each of several test cases, first a line with the test case name, then N lines of M columns, where N is the number of rows and M the number of columns, and then a single line containing just '.'.

The characters of the N lines of M columns describe the state of the board, with one character for each square containing:

```
        X       If the square is occupied by X.
        O       If the square is occupied by O.
        G       If the square is occupied by a gorilla.
        .       If the square is unoccupied.
```

No input line is longer than 80 characters. Input ends with an end of file.

Output

For each test case, an exact copy of the input, but with the '.'s replaced by one of the following indicators of what will happen if P moves to the '.'ed square and then both players play optimally:

```
        W       if P will win
        L       if P will lose
        T       if the game will tie
```

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

-- SAMPLE 1 -
...
.XO
...
.
-- SAMPLE 2 --
...G.
.OX.G
.....
.
-- SAMPLE 3 --
...GO
.GXOG
...XG
.


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

-- SAMPLE 1 -
WWW
TXO
WWW
.
-- SAMPLE 2 --
TWTGL
LOXLG
TLTLL
.
-- SAMPLE 3 --
LWLGO
LGXOG
LLWXG
.
```

```
File:       gorillatoe.txt
Author:     Bob Walton <walton@seas.harvard.edu>
Date:       Sat Sep  1 08:12:28 EDT 2012

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: 2012/09/01 12:16:39 $
    $RCSfile: gorillatoe.txt,v $
    $Revision: 1.4 $
```

```
Angle Puzzle
----- ------

An angle puzzle consists of a finite set of vertices in
the plane and a set of equations of the form

        xyz = some angle
    or
        xyz = ?

where x, y, and z are vertices and x != y != z != x.
xyz is interpreted as the angle at vertex y in the
triangle with vertices x, y, and z, if x, y, and z are
not all on the same infinite line, with this angle being
positive if traversing from x to y to z goes in the
counter-clockwise direction about the triangle, and
negative if clockwise.

But if x, y, and z are on the same infinite line, xyz =
0 if x and z are on the same side of y, and xyz = 180
if x and z are on opposite sides of y.  These are useful
ways of saying that x, y, and z are on the same line.

Note that xyz = - zyx always and adding multiples of
360 to an angle does not change the angle (so 180 and
-180 are equal as angles).  All input and output angles
are measured in degrees and are in the range (-180,180],
so +180 is allowed for input/output but -180 is NOT
allowed and must be replaced by +180.

The puzzle requires you to solve for the ?'s in the
the equations.

Sample 1 below is actually solvable using elementary
geometry without trigonometry, but in general you will
need to use trigonometry to solve these puzzles, as is
done in sample 2.


Input
-----

For each of several test cases, first a line with the
test case name, and then a sequence of lines with equa-
tions as above, and then a line containing just '.'.
The vertex names are all single capital letters.  The
angles are all in degrees.  The only space characters
in any input line other than the test case name line
are the two surrounding the '='.  No line is longer
than 80 characters.

No two vertices with different names are identical.

Input ends with an end of file.



Output
------

For each test case, a copy of the input but with ALL
'?'s replaced by numbers.  All output angles should have
exactly 3 decimal places and be in the range
(-180,+180].  The output should be an exact copy of the
input except for the replacement of '?'s by the numbers
and the rounding of input angles to 3 decimal places.

This problem is actually open ended in that we do not
expect you to find all the angles that might be
determined from the given input.  But you must find the
angles you are asked to find.  These can be found by
using only non-trigonometric constraints on angles plus
trigonometrically computed relative positions of the
vertices of any triangle two of whose angles are known.
```

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

-- SAMPLE 1 --
ABC = 60.000000000
BCA = 60.000000000
DBC = 30.000000000
ADC = 180
DAB = ?
ADB = ?
.
-- SAMPLE 2 --
ABD = 60.000000000
DBC = 20.000000000
ADC = 180.000000000
EAB = 70.000000000
CAE = 10.000000000
CEB = 180.000000000
AED = ?
AEB = ?
EDB = ?
CBE = ?
.
```

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

-- SAMPLE 1 --
ABC = 60.000
BCA = 60.000
DBC = 30.000
ADC = 180.000
DAB = 60.000
ADB = -90.000
.
-- SAMPLE 2 --
ABD = 60.000
DBC = 20.000
ADC = 180.000
EAB = 70.000
CAE = 10.000
CEB = 180.000
AED = 20.000
AEB = -30.000
EDB = 110.000
CBE = 0.000
.
```

File: anglepuzzle.txt
Author: Bob Walton <walton@seas.harvard.edu>
Date: Wed Oct 10 02:54:56 EDT 2012

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

    $Author: walton $
    $Date: 2012/10/10 07:18:45 $
    $RCSfile: anglepuzzle.txt,v $
    $Revision: 1.11 $

Broppers

On the planet of Pons, the Xflea (rough translation: bridge hoppers, or colloquially, broppers) hold a race/ contest every Xflit (rough translation, 5 years, 8 months).

The course consists of a isles connected by bridges. There is a start isle and a finish isle. A team consists of broppers who are all initially at the start isle and who try to get to the finish isle. Their movements are controlled by a gong; when the gong sounds, each bropper either stays put or crosses a bridge. The team goal is to get as many team members as possible to the finish. Being broppers, a team is of unbounded size, and the team has an unbounded amount of time to get from start to finish.

Wait, somethings missing. We forgot to tell you about the bridges. Only one bropper can cross a bridge at a time (i.e., between one gong and the next gong), and the bridges are all 1-way. There are two kinds of bridges, F-bridges and S-bridges. An F-bridge, or falling bridge, falls down immediately after it is crossed, and cannot be used a second time. However S-bridges, or synchronized bridges, are another kettle of brop altogether. S-bridges do NOT fall down. But they can only be used if all S-bridges are crossed (by different broppers) simultaneously. That is, if at a gong some bropper tries to cross every S-bridge, they all succeed, but if some S-bridge has no bropper trying to cross it, none of the broppers trying to cross S-bridges go anywhere. Also, there no way for broppers to get from the end of any S-bridge to the beginning of some S-bridge, so a bropper cannot cross an S-bridge and then go to the beginning of some S-bridge to help other broppers across.

The layout of the race/contest is different for every team. But this is not as unfair as it seems, as the course is always very large and random, and teams never get nearly as many members to the finish as they could in theory. Just to prove this, a team is told in advance the maximum number of members they could get from start to finish. And you have been hired to write a program which will compute this so the contest can be properly run.

Input

For each of several test cases, one line containing the test case name, followed by lines that describe bridges. These have the form

```
        type begin end
```

where type is ‘F’ or ‘S’ and begin and end are isle numbers. After these lines is a line containing just ‘.’.

Isles are numbered 1, 2, 3, ..., N for some N <= 10,000. The start isle is always isle 1 and the finish is always isle 2. No S-bridge begins or ends at the start or finish isles, and no two S-bridges start at the same isle or finish at the same isle. There is at most one bridge with a given beginning isle and end isle (but there can be two bridges going in opposite directions between two isles). There are no more than 1,000,000 bridges, and no input line is longer than 80 characters.

Input ends with an end of file.

```
Output
------

For each test case, a copy of the test case name line,
followed by a line containing just the maximum number of
team members who can make it from start to finish.  Note
that a bropper who starts is not required to finish, but
may drop out at any time (e.g., after crossing an
S-bridge).


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

-- SAMPLE 1 --
F 1 3
F 1 5
F 5 2
S 3 4
S 5 6
.
-- SAMPLE 2 --
F 1 3
F 1 5
F 5 2
F 4 2
S 3 4
S 5 6
.
-- SAMPLE 3 --
F 1 3
F 1 5
F 5 2
F 4 2
F 6 2
S 3 4
S 5 6
.

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

-- SAMPLE 1 --
1
-- SAMPLE 2 --
1
-- SAMPLE 3 --
2
```

Thanks: To the books of Iain M. Banks for stylistic
        guidance.

File: broppers.txt
Author: Bob Walton <walton@seas.harvard.edu>
Date: Wed Oct 10 03:20:56 EDT 2012

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

    $Author: walton $
    $Date: 2012/10/10 07:23:26 $
    $RCSfile: broppers.txt,v $
    $Revision: 1.10 $