Problems Index Thu Oct 15 07:57:01 AM EDT 2009

BOSPRE 2009 PROBLEMS
------ ---- --------

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

```
    problems/turtledraw
        The Art of Turtle
        Boston Preliminary 2009

    problems/evensteven
        Why is this named Even-Steven?
        Boston Preliminary 2009

    problems/nasheq
        Does math really help?
        Boston Preliminary 2009

    problems/myrsync
        Clever summing saves cycles.
        Boston Preliminary 2009

    problems/combinators
        Functions without variables.
        Boston Preliminary 2009

    problems/logistic
        Statistical stability can be weird.
        Boston Preliminary 2009
```

Turtle Draw

A beaver, a dog, a frog, and a man were sharing a corner of a pond on a sweltering evening in early August. The beaver was considering air conditioning, the frog was imagining a water fall, the dog was happy to just swim after tennis balls, and the man knew he needed to write a computer program.

When he got home the man wrote a program called ‘turtle-draw’, in honor of the turtle that lived in the pond. She was not with the foursome that particular August evening, which just as well, as a 40 pound snapping turtle is a bit of a pond party pooper.

Input

The input contains a series of commands for an imaginary turtle living on an infinite board of squares. At any time, the turtle is on a particular square, and is facing in one of four directions, up, right, down, or left. In the beginning the turtle is facing up and all squares are blank.

The commands are:

```
M           Move forward one square.
L           Turn left 90 degrees.
R           Turn right 90 degrees.
<other>     Any other non-blank character:
            write the character on the
            current square and THEN move
            forward one square.
```

The input is a sequence of test cases. Each test case begins with a line that names the test case. This is followed by one or more lines which contain commands for the turtle. No command line contains whitespace characters, and no command line contains just the character ‘.’. The test case ends with a line containing just ‘.’ (exactly one ‘.’).

In any test case the turtle will not wander more than 100 squares in any direction away from its starting position. No input line will contain more than 80 characters.

Output

For each test case, first one line that is an exact copy of the test case name line, then a single empty line (with no characters), and then just the portion of the infinite board that contains non-blank squares. Specifically, this portion of the board should NOT have any blank lines at its top or bottom, or any blank columns at its left or right edges. At the end of the test case, right after the portion of the board with no blank lines, there should be a single blank line.

Thus the output for each test case should have exactly two blank lines: the second line (after the name and before the board), and the last line (after the board). The entire output for ALL test cases ends with a blank line (if you get a ‘format error’ score you may have the blank lines wrong).

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

--SIGN--
EWRMGORMDOWNLMTHENLMPU
.
--HAT--
L/_M____M_\LL
MMML|RM____RM|
.
--DOG--
RR***LMR***LMR***L**RML****LMR**L****RML***RML***
L//////_\\\\\\L\**RML****
LLMRMMMMMMMMMMMMMMMML****LMR**
LMMMMMMMMMMLMMMML--MMM--
.
```

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

--SIGN--

 GO
W  D
E  O
   W    U
   N    P
    THEN

--HAT--

   ____
\_|____|_/

--DOG--

 \\\\\\_//////
 **           **
 **           **
* *           * *
*  * --   -- *  *
*  *         *  *
*  *         *  *
    *       *
    *       *
    *       *
    **    ***
      ****

NOTE: This output ends with a single blank line.
```

```
File:      turtledraw.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Thu Oct 15 05:42:30 EDT 2009

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: 2009/10/15 09:44:48 $
    $RCSfile: turtledraw.txt,v $
    $Revision: 1.5 $
```

```
Even Steven
-----------

Even-Steven is a one person card game played as follows.

First you deal yourself a hand of cards.  Then you deal
cards one at a time, and every time you deal a card,
you must cover it by playing one card of equal or higher
value from your hand, or you lose the game.  If your
hand runs out of cards without losing, you win.


Programs
--------

You are asked to write a program called ‘evensteven’
that plays your hand.  A program called ‘dealer’ is
provided that is the dealer.  Each of the two programs
reads its standard input and writes its standard
output.  The standard input and output of the dealer
is connected to a terminal or file.  The standard input
and output of your ‘evensteven’ program is connected to
the dealer program: the dealer writes what ‘evensteven’
reads and reads what ‘evensteven’ writes.

If you invoke the programs with the command

        dealer evensteven

the dealer starts your ‘evensteven’ program as a
subprogram (actually subprocess) of the dealer.  Then
the dealer reads from the terminal, writes to your
‘evensteven’ program, reads from your ‘evensteven’
program, and writes to the terminal.

Input for the ‘dealer’ Program
----- --- --- -------- -------

For each of several games, just one line that contains

        N seed

where N is the number of cards dealt to the player’s
hand (0 < N <= 13), and seed is the seed of a pseudo-
random number generator that is used to shuffle the
deck ( 0 < seed < 2147483647 ).

The input ends with an end of file.


Input for ‘evensteven’ Program
----- --- ------------ -------

For each of several games, first a line that describes
your hand (the first N cards of the shuffled deck).
This line has the form

        N C1 C2 ... CN

where N is the number of cards and each Ci is a card
specified in the format:

        Ci := <value><suit>

        <value> := 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
                 | J | Q | K | A

        <suit> := c | d | h | s

Here the <value> is a number or J for ‘jack’, Q for
‘queen’, K for ‘king’, or A for ‘ace’.  A <value> is
greater if it is later in the above list; i.e., the
highest value is ‘A’ for ‘ace’ and the lowest is ‘2’.
```

The <suit> is c for ‘clubs’, d for ‘diamonds’, h for ‘hearts’, or s for ‘spades’. The suit of a card has NO affect on the card’s value.

So ‘2c’ is the two of clubs and ‘Jh’ the jack of hearts.

After this first line, each following line names a single card being dealt from the reminder of the deck, or is one of the following:

```
        YOU WIN
        YOU LOSE
```

Either of these last two lines ends the game.

When you read an end of file, you MUST terminate the program (ALL the games are over).

Output for ‘evensteven’ Program
------ --- ------------ -------

For each game, every time you read a line naming a single card dealt after your hand has been dealt, you must output one line naming the card in your hand that you are playing. Once you play a card this way, it is removed from your hand and you cannot play it again .

A played (output) card is non-losing if its value is not less than the value of the dealt (input) card.

You MUST play a card for each card dealt, even if you have only a losing card (in which case the next line you read will be ‘YOU LOSE’). Otherwise the dealer will time out waiting for you to output a card, and you will be given a ‘program crashed’ score as detailed under ‘Scoring’ below. Some hands are necessarily losing, and you must lose these gracefully to have a successful ‘evensteven’ program.

WARNING: If you are programming in C you must execute

```
        fflush (stdio);
```

after writing each line to the standard output, or your output will be trapped in a buffer and never get to the dealer. In C++ the ‘endl’ IO manipulator flushes the buffer and in JAVA ‘println’ flushes the buffer, so nothing unusual needs to be done for these languages.

Output for the ‘dealer’ Program
------ --- --- -------- -------

For each of the several games, first a line containing ‘Game #’ where # = 1, 2, 3, ... is the number of the game, and then one of the lines:

```
        YOU WIN
        YOU LOSE NECESSARILY
        YOU LOSE UNNECESSARILY
```

You can lose in either of two ways: ‘NECESSARILY’ because there is no way to play your hand and win, and ‘UNNECESSARILY’ because you played a winning hand badly (the dealer is a smart alec).

Scoring

The judge’s test input and output are for the dealer program, and the judge’s test output does NOT contain any ‘YOU LOSE UNNECESSARILY’ lines. Thus if your program loses unnecessarily, you will get a score of ‘incorrect output’.

If your program outputs a badly formatted line the ‘dealer’ program will output an error message and your program will get the score ‘incorrect output’.

Similarly you will get ‘incorrect output’ if you play a card not in your hand or you play the same card twice in a game.

If your program hangs up reading when the dealer is trying to read from your program a line your program failed to write, the dealer will detect this eventually and abort, causing a score of ‘program crashed’.

Debugging

If your ‘evensteven’ program outputs a line that begins with ‘*’, the ‘dealer’ program will copy that line to its standard output and otherwise ignore the line. This can be used for debugging. For example, in C++ one might use

```
        bool debug = false;
        #define dout if ( debug ) cout << "* "
        . . .
        main ( int argc )
        {
            debug = ( argc > 1 );
            . . .
            dout << "my debugging message" << endl;
            . . .
```

The ‘dealer’ program passes extra arguments onto the ‘evensteven’ program; e.g.,

```
        dealer evensteven debug
```

executes ‘evensteven debug’. See the ‘make debug’ command in the ‘solving’ help file.

However, if you output ‘*’ lines when just ‘evensteven’ with no arguments is called, in the code you submit to the judge, these lines will appear in the output file and you will get the score ‘Incorrect Output’.

One use for debugging ‘*’ lines is simply to echo all input to and output from ‘evensteven’ so you can see how a game is going.

Sample Input for Dealer
------ ----- --- ------

```
4 876390176
4 653723903
```

```
Sample Input for Evensteven
------ ----- --- ----------

4 4h 2d Qs Jc
8c
Qc
9s
YOU LOSE
4 Kd 8c As 5s
3s
6h
Kh
5c
YOU WIN


Sample Output for Evensteven
------ ------ --- ----------

Jc
Qs
2d
5s
8c
Kd
As


Sample Output for Dealer
------ ------ --- ------

Game 1
YOU LOSE NECESSARILY
Game 2
YOU WIN
```


File: evensteven.txt
Author: Bob Walton <walton@seas.harvard.edu>
Date: Thu Oct 15 05:56:44 EDT 2009

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

```
    $Author: walton $
    $Date: 2009/10/15 09:58:04 $
    $RCSfile: evensteven.txt,v $
    $Revision: 1.9 $
```

Nash Equilibrium
---- -----------

In Game Theory a simple finite two-person game consists of
- two players
- a finite set of 'strategies' for each player
- a payoff matrix for each player

Let the two players be R ('rows') and C ('columns'). Let the strategies for R be labeled 1, 2, ..., NR, and the strategies for C be labeled 1, 2, ..., NC. The payoff matrix for R is R(r,c) where r is a strategy for R and c a strategy for C; while the payoff matrix for C is similarly C(r,c). These matrices have integer elements and larger payoffs are better.

A round of the game consists of each player privately choosing a strategy, and then the two players simultaneously announce their strategies. Let r be R's strategy and c be C's. Then the round pays R the amount R(r,c) and pays C the amount C(r,c).

You might think this simple minded, but it is an abstraction that covers games such as TicTacToe. For TicTacToe a strategy will be some algorithm for playing the game, and a round will consist of each player choosing a strategy privately and the game being played according to these strategies. The payoffs might be +1 for winning, -1 for losing, and 0 for tie.

The payoff matrices are commonly specified by a single matrix whose elements are 'R(r,c)/C(r,c)'. Thus we have the classic game:

Prisoner's Dilemma

	1=Remain-Silent	2=Confess
1=Remain-Silent	-6/-6	-120/0
2=Confess	0/-120	-60/-60

Two prisoner's are charge with a crime they committed. If both remain silent, they each serve 6 months. If both confess, they each serve 60 months (5 years). If one confesses and the other remains silent, the confessor goes free and the other prisoner serves 120 months (10 years).

This game is symmetric: R(r,c) = C(c,r). But not all games are.

Given a game, a 'dominated strategy' for R is an r such that there exists an r' for which R(r',c) >= R(r,c) for every c and R(r',c') > R(r,c') for some c'. The idea here is that r' is always a better strategy than r, so r is 'dominated' if and only if there is some other strategy that is always better. A dominated strategy for C is defined analogously.

One way to decide how to game should be played is to iteratively eliminate dominated strategies. In the case of Prisoner's Dilemma, Remaining Silent is a dominated strategy for both prisoners, and eliminating it gives a game in which both prisoners Confess.

Given a game, a 'Nash Equilibrium' is a pair (r,c) such that for every r', R(r,c) >= R(r',c) and for every c', C(r,c) >= C(r,c'). The Prisoner's Dilemma has one Nash Equilibrium: the pair where both players Confess.

Given a game you are asked to find all the dominated strategies and all the Nash Equilibria.

Input

The input consists of test cases. Each test case begins with a line containing the name of the test case. This is followed by a single line containing NR and NC in that order, and this is followed by NR lines each containing NC number pairs, where each pair is written as ‘#/#’ where # stands for an integer. The c’th pair of the r’th line is R(r,c)/C(r,c).

1 <= NR,NC <= 20.

Some of the input lines may be very long.

The input is terminated by an end of file.

Output

For each test case, four lines, the first being a copy of the first test case input line that contains the test case name. The remaining three lines, in order, are:

```
    Dominated R Strategies: r1 r2 ...
    Dominated C Strategies: c1 c2 ...
    Nash Equilibria: (r1’,c1’) (r2’,c2’) ...
```

where r1, r2, ..., c1, c2, ... are strategy numbers (integers from 1 through NR and 1 through NC respectively) and (r1’,c1’), (r2’,c2’), ... are strategy pairs.

You must list ALL the dominated R and C strategies and all the Nash Equilibria and not have duplicates, but the order does not matter. Its possible that there will be nothing after a ‘:’ on a line. Some of the output lines will be very long.

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

```
-- PRISONER’S DILEMMA --
2 2
-6/-6    -120/0
0/-120   -60/-60
-- BATTLE OF SEXES --
2 2
0/0 2/1
1/2 0/0
-- MATCHING PENNIES --
2 2
1/-1 -1/1
-1/1 1/-1
-- COURNOT COMPETITION, 3 GOODS, PRICE 5, COST 1
4 4
0/0 0/3 0/4 0/3
3/0 2/2 1/2 0/0
4/0 2/1 0/0 -2/-3
3/0 0/0 -3/-2 -3/-3
```

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

-- PRISONER'S DILEMMA --
Dominated R Strategies: 1
Dominated C Strategies: 1
Nash Equilibria: (2,2)
-- BATTLE OF SEXES --
Dominated R Strategies:
Dominated C Strategies:
Nash Equilibria: (1,2) (2,1)
-- MATCHING PENNIES --
Dominated R Strategies:
Dominated C Strategies:
Nash Equilibria:
-- COURNOT COMPETITION, 3 GOODS, PRICE 5, COST 1
Dominated R Strategies: 1 4
Dominated C Strategies: 1 4
Nash Equilibria: (2,2) (2,3) (3,2)


File:      nasheq.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Sun Oct 11 23:19:29 EDT 2009

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: 2009/10/12 04:14:48 $
    $RCSfile: nasheq.txt,v $
    $Revision: 1.7 $
```

```
My Rsync
-- -----

The UNIX rsync program copies a file F at location L to
a remote location L' which is accessible from L only by
slow communications.  It optimizes the case where an
older version F' of the file already exists at L'.  F'
is divided into disjoint S byte blocks, and the MD5
signatures of these blocks are communicated by L' to L.
Then L sends to L' the bytes of F as follows: if the
next S bytes to be sent match a block of F', the identi-
fier of that block is sent, and otherwise the next byte
is sent.  Here S bytes match a block in F' if both have
the same MD5 signature, which is only 16 bytes, so L'
only has to send 16 bytes for every S bytes of F', and
this is faster than having L' send all of F' if S is
much greater than 16.

However, if we do as we have said, then for every byte
of F the MD5 signature of the S byte block starting at
that byte would have to be computed.  This is too expen-
sive computationally.

So L' reports for every block both its MD5 signature and
a 32-bit rolling checksum.  L uses the rolling checksum
to find blocks in F that might with high probability be
identical to blocks in F', and then computes the MD5
signatures of just those blocks, to check if the blocks
are indeed identical.

What do we mean by a rolling checksum?  We are looking
at the sequence of S byte blocks of F that begin at all
the possible different byte offsets in F.  Suppose we
have a pointer into F and relative to that pointer the
next S + 1 bytes are

        B(0), B(1), B(2), ..., B(S-1), B(S)

An example rolling checksum for the current block is

  b = (B(0) + B(1) + B(2) + ... + B(S-1)) mod 2**16

The value of this checksum for the next block in the
sequence is

  bnext = (B(1) + B(2) + ... + B(S-1) + B(S)) mod 2**16
        = (b + B(S) - B(0)) mod 2**16

That is, bnext can be computed quickly from b and the
byte B(0) we are discarding and the byte B(S) we are
adding to make the next block from the current block.
We call b a `rolling' checksum because bnext can be
computed quickly from b.

Another example of a rolling checksum is

    c = (S*B0 + (S-1)*B1 + (S-2)*B2 + ... + 1*B(S-1))
        mod 2**16

for which

    cnext = (S*B1 + (S-1)*B2 + (S-2)*B3 + ... + 1*B(S))
            mod 2**16
          = (c + bnext - S*B0) mod 2**16

Here we use bnext to help compute cnext.  Lastly, we
can combine these two rolling checksums into one:

    d = b + 2**16 * c

which is the 32-bit rolling checksum that we will use.

Note that in the above a byte is an UNSIGNED 8 bit
integer (an `unsigned char' in C/C++ , and as JAVA
does not have unsigned integer data, you must convert
each byte to an int and then & with 0xFF in JAVA).
```

Input

The standard input consists of test cases. Each test case begins with a line containing the name of the test case. The second line of the test case contains a data file name (the name of F), and the third line contains the block size. The lines following this each describe one block of the remote file F’, and each of these lines holds an MD5 signature followed by a single space followed by a rolling checksum. The signature is 32 hexadecimal digits (0, ..., 9, A, ..., F), and the rolling checksum is 8 hexadecimal digits. The last line of the test case contains just ‘.’, which signals the end of the test case.

The input file name will not contain any white-space characters, and the block size will be a decimal number. No standard input line will be longer than 80 characters. The standard input will be terminated by an end of file after the last test case.

You must open each input file F for reading, and NOT for writing. If you open it for reading and writing, your program may fail, and WORSE, it might work when you test it and then fail when the judge tests it because when the judge runs it your program will not be allowed to open files for writing.

Output

For each test case, first output an exact copy of the first three lines of the test case: the test case name, the file name, and the block size. Then for each offset in file F of a block whose rolling checksum matches the rolling checksum of some block of F’, output the line

```
        offset block-number
```

where block-number is the block number of the block of F’ whose MD5 sum matches that of the block of F at the given offset, or is -1 if there is none. These lines must be in order of increasing offset. The blocks of F’ are numbered 0, 1, 2 ... .

Lastly output a line containing just ‘.’ to end the test case output.

Notes

To compute an MD5 sum of an S byte block:

In C:

```
#include <openssl/md5.h>
unsigned char signature[16];
unsigned char block[S];
... read block ...
MD5( block, S, signature );
```

In C++:

```
extern "C" {
#include <openssl/md5.h>
}
unsigned char signature[16];
unsigned char block[S];
... read block ...
MD5( block, S, signature );
```

In JAVA:

```
import java.security.*;

static byte[] MD5 ( byte[] block )
    throws NoSuchAlgorithmException
{
     MessageDigest md =
         MessageDigest.getInstance ( "MD5" );
     return md.digest ( block );
}
```

Here the MD5 sum is called a signature and is represented as a 16 byte string, where each byte represents 2 hexadecimal digits, with the first byte representing the highest order digits.

If you use gcc or g++ directly (instead of using the Makefile you are provided) you need to use the -lssl library option.

On modern computers computation of MD5 sums is so fast compared to input/output CPU time that we were unable to construct sensible test cases where the optimization of using rolling sums to reduce the amount of MD5 sum computation actually made a large difference in CPU time.

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

-- SAMPLE 1 --
sample1.dat
256
0665A333D10B4F10495EDCD35E8F2904 94127CB5
7CB5FA5E037EFF272462C92867AFC1B9 BE387EB4
D94B841EB5F4C528ACFFA4D2BD068503 94127CB5
.
-- SAMPLE 2 --
sample2.dat
4096
B26EBEE4CB66A9CC513F48293E676CA9 FB23138E
6167E71B305291265C85B37F758DB1BB D217FCA6
87AC778780609BDF81C7E5C144BE48EA 48C909AF
.
```

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

-- SAMPLE 1 --
0 0
256 1
512 2
.
-- SAMPLE 2 --
0 0
8314 2
.
```

```
File:       myrsync.txt
Author:     Bob Walton <walton@seas.harvard.edu>
Date:       Thu Oct 15 06:26:44 EDT 2009

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: 2009/10/15 10:29:49 $
    $RCSfile: myrsync.txt,v $
    $Revision: 1.13 $
```

```
Combinators
-----------

The lambda calculus is a means of representing
functions by means of 'lambda-expressions' that have the
syntax

        lambda-exp ::= variable
                    | ( lambda-exp lambda-exp )
                    | ( \ variable . lambda-exp )
        variable ::= single lower case letter

For example, (\x.x) represents the identity function
that maps an argument x onto itself.  ((\x.x) y)
represents the application of this function to the
variable y.  It happens that this application can be
reduced as ((\x.x) y) => y.  In general ((\x.M[x]) N) =>
M[N] where M[x] denotes any lambda-expression possibly
containing the variable x, N is any lambda-expression,
and M[N] is M[x] with N substituted for the 'free'
occurrences of x.  However, you will not need to compute
applications in this problem, so we will not get into
details (such as what 'free' means).

Here were use '\' to denote the Greek letter 'lambda'.

The combinatorial calculus is another way of expressing
functions that uses the syntax:

        c-expression ::= variable
                    | ( c-expression c-expression )
                    | K
                    | S

        variable ::= lower case letter

where 'c-expression' is shorthand for 'combinatorial
expression', and K and S are constant functions.  In the
combinatorial calculus application is computed using the
rules

        ((KM)N) => M
        (((SM)N)P) => ((MP)(NP))

for any c-expressions M and N.  These rules are simpler
than the rules for lambda-calculus, in the sense that
there is no need to substitute for variables.

A lambda-expression can be rewritten into an equivalent
combinatorial expression using the following rules:

        (\v.w)          => (Kw)
        (\v.K)          => (KK)
        (\v.S)          => (KS)
        (\v.v)          => ((SK)K)
        (\v.(MN))       => ((S(\v.M))(\v.N))

for any DISTINCT variables v and w and any expressions
M and N.

You are asked to convert lambda-expressions into
c-expressions using this last set of rules.

Notice that you may have to apply these rules to sub-
expressions before you can apply them to containing
expressions.  Thus

   (\x.(\y.x)) => (\x.(Kx)) => ((S(\x.K))(\x.x))
               => ((S(KK))((SK)K))
```

Input

The input consists of test cases. Each test case begins with a line containing the name of the test case, and this is followed by a single line containing a lambda-expression. There are no spaces in the lambda-expression line, and no input test case line is longer than 80 characters. The input is terminated by an end of file.

Output

For each test case, three lines, the first two being copies of the two test case input lines, and the third containing the equivalent c-expression, as computed by the above conversion rules. The last line may be very, very long.

Note that both input and output are fully parenthesized; there are NO implicit parentheses in either. Also there are no whitespace characters inside expressions.

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

```
-- IDENTITY --
(\x.x)
-- APPLICATION --
(\x.(\y.(xy)))
-- K --
(\x.(\y.x))
```

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

```
-- IDENTITY --
(\x.x)
((SK)K)
-- APPLICATION --
(\x.(\y.(xy)))
((S((S(KS))((S(KK))((SK)K))))((S((S(KS))(KK)))(KK)))
-- K --
(\x.(\y.x))
((S(KK))((SK)K))
```

File: combinators.txt
Author: Bob Walton <walton@seas.harvard.edu>
Date: Thu Oct 15 06:30:59 EDT 2009

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

```
    $Author: walton $
    $Date: 2009/10/15 10:31:17 $
    $RCSfile: combinators.txt,v $
    $Revision: 1.10 $
```

Logistic Population Growth

Logistic population growth is given by the equation

$$dN/dt = rN(1 - N/K)$$

where

- t is the time
- N is the current population (a function of t)
- r is the population growth rate, a constant
- K is the carrying capacity of the environment, a constant (we allow this to be non-integral)

The population starts from an initial value N(0) at t = 0 and grows or shrinks exponentially until it approximately equals K.

The above equation has the solution

$$N(t) = K / (1 + F * \exp (-rt))$$

where

$$F = (K - N(0)) / N(0)$$

These equations define what is called a ‘deterministic model’. Real population growth curves tend to wander from deterministic model predictions unless N is large, as we will show in this problem.

Population growth can be more carefully modeled by a ‘stochastic model’. To build this model we define

$$B(N) = \max (0, (b0 - b1 * N) * N) \qquad \text{Birth rate}$$
$$D(N) = (d0 + d1 * N) * N \qquad \text{Death rate}$$

p(N,t) probability that the population is N at time t

b0, b1, d0, d1 >= 0 are constants of the model

From this we get the stochastic differential equation

$$dp(N,t)/dt = - p(N,t)*(B(N) + D(N)) + p(N+1,t)*D(N+1) + p(N-1,t)*B(N-1)$$

Here the corresponding deterministic model is

$$dN/dt = B(N) - D(N) = ((b0 - d0) - (b1 + d1) * N) * N$$

so comparing to the above we get

$$r = b0 - d0$$
$$r/K = b1 + d1$$

hence

$$K = (b0 - d0) / (b1 + d1)$$

For the stochastic model all four constants b0, b1, d0, d1 are needed to define the model, but for the deterministic model only r and K are needed.

The deterministic initial condition N = N(0) is equivalent to the stochastic initial condition p(N,0) = 1 and p(n,0) = 0 for n != N.

Integrating the stochastic differential equation to find p(N,t) turns out to be difficult, even numerically using a computer. But there is a steady state where all the dp(N,t)/dt = 0, and we can solve for the p(N,t) in this steady state. Call these steady state probabilities P(N), and then replacing dp(N,t)/dt by 0 and p(N,t) by P(N) in the above equation we get

$$0 = - P(N)*(B(N) + D(N)) + P(N+1)*D(N+1) + P(N-1)*B(N-1)$$

This equation in turn can be rewritten as

```
  P(N+1)*D(N+1) - P(N)*B(N) = P(N)*D(N) - P(N-1)*B(N-1)
    = G a constant independent of N
```

For N = 0 this equation is

```
        P(0)*D(0) - P(-1)*B(-1) = 0
```

and if D(0) = P(-1) = 0 we have G = 0. Thus

```
        P(N+1) = P(N)*B(N)/D(N+1) if D(N+1) != 0
```

We have B(0) = 0 and D(N) > 0 for N >= 1 and this gives the solution P(0) = 1, P(N) = 0 for N > 0, which is the ‘extinction solution’ and is not very interes- ting.

However, if we assume that extinction never actually occurs, this is equivalent to assuming that P(0) = 0 and throwing out the equation

```
        P(0)*D(0) - P(-1)*B(-1) = G
```

Then we have

```
        P(1)*D(1) - P(0)*B(0) = G
```

and as P(0)= 0, G = P(1)D(1). We thus get

```
        P(N+1) = ( P(N)*B(N) + P(1)*D(1) )/D(N+1)
```

and if we set

```
        Q(1) = 1

        Q(N+1) = ( Q(N)*B(N) + Q(1)*D(1) )/D(N+1)
```

then

```
        P(N) = Q(N)/(sum Q(N) for all N >= 1)
```

which normalizes P(N) so the sum of the probabilities is 1.

Note that to actually sum the Q(N) in a computer you need to stop summing at some finite value of N. In our case B(N) = 0 when N >= b0/b1, and

```
        sum ( Q(1)/D(N) for N > L ) <= Q(1)/(d1*L)
```

(because (sum 1/N**2 for N > L) is <= 1/L), so if we set

```
        L = max (b0/b1,  1/d1))
```

the sum of Q(N) for N > L will be at most Q(1), which we expect to be very small part of the total sum, so we restrict the summing to 1 <= N <= L and our equation becomes

```
        P(N) = Q(N)/(sum Q(N) for 1 <= N <= L )
```

Given this we define the steady state statistics of the model as

```
    MEAN = steady state mean of N
            = sum( N * P(N) for 1 <= N <= L )
    VAR  = steady state variance of N
           = sum( (N-MEAN)**2 * P(N) for 1 <= N <= L )
    STD  = steady state standard deviation of N
           = sqrt ( VAR )
```

Lastly a simulation of the stochastic model can be implemented by the following pseudo-code

```
    N = N(0)
    t = 0
    loop:
        choose time s to next event
        choose whether next event is birth or death
        if next event is birth: N = N + 1
        else if next event is death: N = N - 1
        t = t + s
```

Events occur at the rate R(N) = B(N)+D(N) so s is an exponentially distributed random variable such that probability {s’ => 0 : s’ <= s} = exp (- R(N) * s). Note that this equals

```
    probability {s’ => 0 : exp(-R(N)*s’) >= exp(-R(N)*s}
```

so if we set Y = exp (- R(N) * s) we get

```
    probability {s’ => 0 : exp(-R(N)*s’) >= Y} = Y
```

and Y = exp(-R(N)*s) is therefore uniformly distributed. So we can choose s by

```
    to choose s:
        pick a pseudo-random uniformly distributed
             number Y, 0 <= Y <= 1.
        set s = - (ln Y)/R(N)
```

The relative probabilities of births and deaths are B(N) and D(N) so

```
    to choose whether the next event is a birth or
    a death:
        pick a pseudo-random uniformly distributed
             number Y, 0 <= Y <= 1.
        if Y <= B(N)/(B(N)+D(N)) the event is a birth
        otherwise it is a death
```

If N == 0, then B(0) = D(0) = R(0) = 0, and this is a special case in which N is stuck at 0 forever.

By now you must have guessed that you are going to be asked to compute all the above. Furthermore, you must get very precisely the same answers as the judge. To do this you need to use double precision floating point numbers and the following pseudo-random number generator:

```
    C or C++:

        long long seed;
        double random_Y ( void )
        {
            seed = 16807 * seed;
            seed = seed % 2147483647
            return double(seed) / 2147483646;
        }

    JAVA:
        long seed;
        double random_Y ( void )
        {
            seed = 16807 * seed;
            seed = seed % 2147483647
            return double(seed) / 2147483646;
        }
```

You will be given an initial value of seed, and for each pseudo-random number Y you need (including the first), you call random_Y().

Input

For each of several test case, first a line containing just the name of the test case. Then a line containing

```
    b0 b1 d0 d1 N(0) Tsize Nsize Tinc Ninc

where b0 .. N(0) define the model, Tsize is the number
of lines in the plot to be produced below, Nsize is
the number of columns in each of these lines, Tinc is
the amount time is incremented between these lines,
and Ninc is the amount N is incremented between columns
of these lines.

        b0, b1, d0, d1, Tinc, Ninc are floating point

        N(0), Tsize, Nsize are integers

        0 <= b0, b1, d0, d1

        0 < N(0)

        0 < Tsize <= 100

        0 < Nsize <= 80

        0 < Tinc

        0 < Ninc

The lines of a test case between the second line of
the test case and the last line of the test case
each have the form

    C seed

where C is the display character for plotting (see
‘Output’ below) and seed is a pseudo-random number
generator seed:

        0 < seed < 2147483647 ( == 2**31-1 )

The last line of a test case contains just ‘.’.

Input ends with an end of file.


Output
------

For each test case, first output an exact copy of the
first test case input line which names the test case.

Then output a plot containing Tsize lines each with
Nsize columns.  The T+1’st line corresponds to the
time t = T*Tinc (so the first line corresponds to
t = 0).   To plot a number x with a display character
C on a line, put the character C in column

        round ( x / Ninc ) + 1

where ‘round’ rounds to the nearest integer.

For each seed you are to plot the simulation using
that seed with the display character given on the
same input line as the seed.  Note that display
characters from a simulation may overwrite display
characters from a previous simulation.

Then you are to plot the deterministic model N(t)
using the display character ‘*’.  Note that this
display character may overwrite simulation display
characters.

There should be NO TABs in any plot line.  You may end
a plot line with single space characters.

After the plot output the line:

    r = #, K = #, MEAN = #, STD = #

where the #’s are as follows:
```

```
    r = b0 - d0 is the rate of population growth when
        N is small

    K = (b0 - d0) / ( b1 + d1 ) is the carrying capacity

    MEAN = mean of the stochastic steady state as
           computed above

    STD  = standard deviation of the stochastic steady
           state as computed above
```

Print all the #’s to at least 2 decimal places.

Note the spacing required in this line: ‘=’s are surrounded by whitespace and ‘,’s are followed by whitespace but NOT preceded by whitespace. Also, you may NOT use TABs in this line. Failure to observe these rules may result in a ‘format error’ score.

Note that as decisions requiring comparison of floating point values are made, output in general will be sensitive to floating point accuracy. However, the judge has tuned the judging input so if you use double precision and follow the above formulae exactly, you will get exactly the plot the judge gets, and you are in fact required to do so. One thing to be careful of is the order in which you use the values returned by random_Y(); specifically you must choose s BEFORE you choose whether the event is a birth or death.

Note that a ‘format error’ score might mean that you have plotted display characters in the wrong columns but have somehow managed to get the right display character overlays, so you may consistently be a column off.

Notes

The simulations reveal that after a population reaches capacity it wanders enough that it does not appear stable. This is because the standard deviation of the stochastic stable solution is not that small.

Therefore

```
                deterministic stability
                          !=
              apparent stochastic stability
```

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

```
-- SAMPLE 1 --
2.2 0.2 0.1 0.1 1 15 50 0.5 0.25
x 838765873
# 098763498
@ 162738493
.
-- SAMPLE 2 --
10.1 0.1 0.0 0.1 10 15 50 0.10 1.2
x 898765873
# 098763498
@ 62738493
.
```

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

-- SAMPLE 1 --
    *
        x*  @
                *   @
                      * x           @
                          *     @   #
                        #  *@       x
                        x   *   #
                            *   x       #
                        @   *   #
                #           *   @   x
        #                   *       @
            #               *   @       x
                        #   *   @   x
                    #   x   *   @
                    #   @   *
r = 2.10, K = 7.00, MEAN = 6.54, STD = 1.75
-- SAMPLE 2 --
        *
         x       *   #
               x  @        *    #
                          x      @ *
                                 x @   *
                                 x   #   * @
                                        x#*@
                                      #   *   @
                                     x    *@
                                     # @  *
                                    #    x*
                                   @ x    *     #
                                        x *@#
                                      x @ *     #
                                          *x @ #
r = 10.10, K = 50.50, MEAN = 49.99, STD = 5.05
```

File: logistic.txt
Author: Bob Walton <walton@seas.harvard.edu>
Date: Wed Oct 14 20:50:13 EDT 2009

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

    $Author: walton $
    $Date: 2009/10/15 00:51:52 $
    $RCSfile: logistic.txt,v $
    $Revision: 1.16 $