PROBABILITY PROBLEMS   Fri Apr 1 05:00:15 EDT 2016

```
    pseudopi
        Throwing chalk.
        Boston Preliminary 2005.

    birthday
        Pseudo-random cannot be really random.
        Boston Preliminary 2008

    pagerank
        Whats popular these days?
        Boston Preliminary 2012

    betterbias
        The Better Bias Bureau improves biased dice.
        Boston Preliminary 2002.

    trends
        Luck or unluck tis common enough.
        Boston Preliminary 2015

    markov
        What is the future of my state?
        Boston Preliminary 2014
```

Pseudo-Random Computation of PI

One of the classic demonstrations of probability is the following. The professor draws a large square on the blackboard, and draws its inscribed circle. Then standing with her back to the board, she throws pieces of chalk at the square. After this she counts the number M of hits in the circle and the number N >= M of hits in the square (including those in the circle), and demonstrates that M/N is about PI/4. This is because PI/4 is the area of the inscribed circle divided by the area of the square, and the probability of hitting any small part of the square is roughly identical to hitting any other small part of the square.

You have been asked to simulate the demonstration in the computer. The square is to be simulated by the unit square in the XY-plane, [0,1]x[0,1], which has (0,0) as its lower left corner and (1,1) as its upper right corner. To simulate throwing the chalk, two random integers X and Y are ‘drawn uniformly’ (see below for details) from the range 0 .. S-1, where S > 0 is some integer. Then the coordinates where the chalk strikes are set at ((X + 0.5)/S, (Y + 0.5)/S). These are inside the square, so all our ‘throws’ count toward N. They are inside the circle, and count toward M, if and only if the chalk strikes at a distance of 0.5 or less from the center of the circle, (0.5, 0.5).

Thus if S = 100 and the first two random integers drawn are 37 and 69, the chalk point is (0.375,0.695) which is distance 0.23 from (0.5,0.5), and is therefore in the circle and counts toward both M and N.

Drawing Random Numbers
------- ------ -------

You are asked to draw pseudo-random numbers according to the equation:

```
        RANDOM = ( RANDOM * MULTIPLIER ) mod MODULUS
```

where RANDOM is the value of the pseudo-random number, the equation steps from the the last pseudo-random number to the next pseudo-random number, and MULTIPLIER and MODULUS are fixed values that determine the pseudo-random number sequence.

To get started, RANDOM is initialized to a value called SEED. The first pseudo-random number in the sequence is not SEED, but the first number after SEED in the sequence.

If MULTIPLIER and MODULUS have good values for this purpose, the resulting sequence of numbers appears when tested to be truly random and uniformly distributed in the range from 1 through MODULUS - 1. Uniformly distributed means all values in this range are equally probable. The choices

```
        MULTIPLIER = 7**5 = 16807
        MODULUS = 2**31 - 1 = 2147483647
```

are very good for this purpose.

For example, if MULTIPLIER and MODULUS are as just given, and the SEED is 374332679, then the first two random numbers are 1429733890 and 1342962947.

A remaining difficulty is how to convert uniformly distributed integers from 1 through MODULUS - 1 to uniformly distributed integers from 0 through S-1. An easy solution, which we will adopt, is to set

```
        S = MODULUS - 1

and subtract 1 from each value of RANDOM.  Thus ‘a chalk
throw’ is simulated by executing

    RANDOM = ( MULTIPLIER * RANDOM ) mod MODULUS
    X = RANDOM - 1
    X = (X + 0.5 ) / S
    RANDOM = ( MULTIPLIER * RANDOM ) mod MODULUS
    Y = RANDOM - 1
    Y = (Y + 0.5 ) / S

to yield (X,Y) in the unit square.

Implementation of the above algorithm requires integers
longer than 32 bits.  In C or C++ you can use doubles
and the fmod function.  Or you can use ‘long long’s and
the % operator.  In JAVA you can use ‘long’s and the %
operator.  Remember, ‘long’s are only 32 bits in C and
C++, but are 64 bits in JAVA.  ‘long long’s are 64 bits
in C and C++.

Input
-----

For each of several test cases, one line containing
four numbers in the order:

        N MULTIPLIER MODULUS SEED

The numbers may be separated by spaces or tabs.  All
input numbers are positive integers below 2**31 (but
some products computed by intermediate computations will
be larger).

Input ends with an end of file.

The simulation is to be done with RANDOM initialized to
SEED (SEED is NOT the first pseudo-random number) and
S = MODULUS - 1.


Output
------

For each test case one line containing five numbers in
the order:

        N MULTIPLIER MODULUS SEED PI_ESTIMATE

where the first four numbers are copied from the input,
and PI_ESTIMATE equals 4*M/N expressed as a decimal
number with exactly 5 decimal places.

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

100      16807 2147483647 374332679
1000     16807 2147483647 374332679
10000    16807 2147483647 374332679
100000   16807 2147483647 374332679
1000000  16807 2147483647 374332679


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

100 16807 2147483647 374332679 3.20000
1000 16807 2147483647 374332679 3.13600
10000 16807 2147483647 374332679 3.15960
100000 16807 2147483647 374332679 3.13888
1000000 16807 2147483647 374332679 3.14167
```

File: pseudopi.txt
Author: Bob Walton <walton@seas.harvard.edu>
Date: Wed Oct 19 07:19:20 EDT 2005

```
The Birthday Paradox
--- -------- -------

Suppose you pick m numbers at random, each between 1 and
n.  How large does m have to be before we can expect to
see two numbers that are the same?

The answer is about the square root of 2n, which is
surprisingly small.  For example, if n = 365, the number
of days in the year, m = 28 will do.  One way to pick
numbers from 1 through 365 is to pick people and take
their birthdays.  Thus if you have 28 people in a room,
you can expect two to have the same birthday.  For this
reason the phenomenon we have just described is called
the ‘Birthday Paradox’.

The Birthday Paradox has some surprising applications.
Suppose you have a pseudo-random number generator that
uses the sequence

        x(i+1) = A*x(i)^2 + B*x(i) + C   (modulo n)

        x(0) = D

for some constants n, A, B, C, and D to generate numbers
x(0), x(1), x(2), ... which we hope act like a sequence
of random numbers.  Suppose they really are like random
numbers.  Then by the birthday paradox the sequence of
numbers will start repeating itself after about m =
square root of 2n numbers.  That is, if we find the
smallest m > 0 and i > 0 such that x(i+m) = x(i), then
typical values of m and i are on the order of the square
root of 2n.  Note that given such values, by the nature
of the above equation, x(j+m) = x(j) for all j >= i, and
we say the sequence has a cycle of length m.

The theory here is not rigorous, because the sequence
x(0), x(1), x(2), is not a rigorously random sequence.
In this problem you will be given n, A, B, C, and D and
asked to compute i and m (the start and length of first
cycle) and print these and the square root of n.  Just
to see if the theory works most of the time.


Input
-----

For each test case, one line containing

        n A B C D

Here 2 <= n <= 40,000; 0 <= A,B,C,D < n.  The input
ends with an end of file.

Output
------

For each test case, one line containing

        n A B C D i m r

where r = ceil ( sqrt ( 2 * n ) ) as an integer (ceil is
the ceiling function and sqrt the square root function),
and each integer of the 8 integers is printed right
adjusted in exactly 7 columns.
```

```
Remark
------

We limit n <= 40,000 in order to permit implementations
to use 32 bit integers.  Note, however, that A * x * x
may not fit into 32 bits, though x * x and A * n will.
If you want to use 64 bit integers (‘long long’ in C and
C++ and ‘long’ in JAVA), you can compute with larger
values of x.

If you use a vector of integers of length n the size of
n will still be limited.  But there are simple clever
implementations that do not need a vector and use almost
no memory for any size of n.


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

100 43 23 17 5
199 0 2 0 1
8191 5 2685 0 7
32749 0 1944 0 5


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

    100     43     23     17      5      2      2     15
    199      0      2      0      1      0     99     20
   8191      5   2685      0      7    155    115    128
  32749      0   1944      0      5      0  32748    256
```

```
Remark
------

The pseudo-random number generators that are actually
used pick A, B, C, and D so the shortest cycle is of
length n or n-1.  One has to be smart about picking A,
B, C, and D.  One old but usable set is

    A = 0, B = 7^5, C = 0, D > 0, n = 2^31 - 1

or in other words,

        x(i+1) = 7^5*x(i) modulo (2^31-1)

with D any non-zero value.  Another set that you can
test your program with is

    A = 0, B = 1944, C = 0, D > 0, n = 2^15 - 19

which should have i = 0 and m = n - 1, the maximum
possible cycle length.

The irony is that to get a good random number generator,
the sequence cannot really be random.


Remark
------

Pollard’s rho algorithm makes clever use of cases such
as
        A = 1, B = 0, C = n - 2

to find factors of n for values of n up to 2^256 + 1.
```

File: birthday.txt
Author: Bob Walton <walton@seas.harvard.edu>
Date: Wed Oct 15 09:18:40 EDT 2008

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

Achieving Better Bias

The Better Bias Bureau takes M faced biased dice with known biases and makes from them N faced biased dice with desired biases. More specifically, if you have an M faced biased die with probability pm[j] of throwing face j, for j from 1 though M, but you really want instead an N faced biased die with probability pn[i] of throwing face i, for i from 1 through N, then good old B3 will give you a method of achieving your desires.

The method is this. You are to consider throwing your M faced die an infinite number of times and writing out the resulting faces, f1, f2, f3, ..., as an infinite fraction .f1f2f3... base M. The fraction will be a number, call it F, in the range from 0 through 1. For any number r, 0 <= r <= 1, we can compute the probability that F <= r.

Let rn[i] be defined so that

```
      probability F <= rn[i]
    = probability that an N faced die throw is k for
      some k <= i
    = sum of pn[k] for k from 1 through i
```

Then when you throw a value F with your M faced die, you should declare it to be face i of the N faced die if

```
                  F < rn[i] if i = 1
or
        rn[i-1] < F < rn[i] if 1 < i < N
or
        rn[i-1] < F         if i = N
or
```

This method produces a properly biased N faced die. Here one ignores the cases where F = rn[i] for some i, as the probability of this happening is 0.

The neat thing about all this is that you do not actually have to throw the M faced die an infinite number of times to decide what the value of i is. After some finite number of throws, depending on the value of F, you can stop, knowing the answer i.

Suppose you stop as soon as possible. What is the expected number of throws of the M faced die necessary to produce one throw of the N faced die?

Note:

For technical reasons, relating to the elimination of ambiguity in judging a contest, we use only one inequality if i = 1 or i = N. To be a little more specific, the judge’s test data has been adjusted so that for all cases that must actually be tested, either both the inequalities rn[i-1] < F and F < rn[i] will be true by a margin definitely larger than the accuracy of the computation, or one of these inequalities will be false by a margin definitely larger than the accuracy of the computation. But it is NOT possible to adjust the judge’s data so the inequalities at the ends, rn[0] = 0.0 < F and F < rn[N] = 1.0, will be true or false by a margin definitely larger than the accuracy of the computation. So we suppress these two inequalities, which we can do without changing the validity of the method.

```
Input
-----

For each of several test cases:

  * M followed by pm[1], pm[2], ..., pm[M] in order.

  * N followed by pn[1], pn[2], ..., pn[N] in order.

Here 2 <= M <= 20, 2 <= N <= 20, and for all j and i
0.01 <= pm[j] <= 0.99, 0.01 <= pn[i] <= 0.99.  The sum
of all the pm[j] equals 1.0, and the sum of all pn[i]
equals 1.0.

Numbers are separated by whitespace, which may consist
of any sequence of spaces, tabs, or newlines.

Input ends with a line containing just a single 0.

You should use double precision floating point arithme-
tic to do input and computation, as more than 6 signifi-
cant figures of accuracy are necessary.


Output
------

For each test case, one line containing the expected
number of throws of the M sided die needed to generate
one throw of the N sided die.  This number should have
exactly 2 decimal places.

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

2 0.5 0.5
4 0.1 0.2
  0.3 0.4
4 0.1 0.2 0.3 0.4
2 0.5 0.5
2 0.5 0.5
3 0.333333333333 0.333333333333 0.333333333333
0


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

3.50
1.45
3.00



Note
----

You might find it interesting to consider at your
leisure whether B3’s method is optimal, i.e., achieves
the minimal expected number of throws of the M faced die
to generate one throw of the N faced die.
```

File: betterbias.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Mon Oct 21 01:22:35 EDT 2002

```
Trends
------

Jack plays a solitary game in which he flips a coin and
gives himself a point on heads but removes a point on
tails.  He starts with a score of zero, and keeps score
as he flips coins.  You would think that his score would
not remain strictly positive for long, or strictly
negative for long, but you would be wrong.  Such long
periods of being strictly positive or strictly negative
are often mis-identified as trends that supposedly prove
the coin is biased.

You have been asked to find the probability of a trend
of length >= m in a sequence of n coin flips.  Here a
'trend' is defined as a sequence of consecutive flips
such that at the end of each flip Jack's score is
strictly positive, or, at the end of each flip Jack's
score is strictly negative.  For comparison purposes,
you are also asked to do this for both unbiased and
biased coins.


Input
-----

For each of several test cases, a line containing

        n m p

where n and m are as above and p is the probability of
heads.

 0 < m,n   m <= n   (m**2)*n <= 50,000,000   0 <= p <= 1

Input ends with an end of file.


Output
------

For each test case, a single line containing

        n m p ptrend

where n, m, p are copied from the input line and ptrend
is the probability of finding a trend of length >= m in
a sequence of n flips.  p and ptrend are to be printed
with exactly 3 decimal places (even if p is input with
fewer decimal places), whereas n and m are integers.


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

10 5 0.5
10 5 0.55
10 8 0.5
10 10 0.5
20 10 0.5
20 10 0.55
20 20 0.5

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

10 5 0.500 0.703
10 5 0.550 0.712
10 8 0.500 0.410
10 10 0.500 0.246
20 10 0.500 0.666
20 10 0.550 0.688
20 20 0.500 0.176
```

File: trends.txt
Author: Bob Walton <walton@seas.harvard.edu>
Date: Thu Aug 13 06:26:42 EDT 2015

```
Markov Recurrence
------ ----------

A finite Markov Chain is a finite set of N states S1,
S2, ..., SN and a matrix p[i,j] of probabilities,
1 <= i,j <= N, such that if the ‘system’ is in state Si,
the next state of the system will be Sj with probability
p[i,j].  It is required that

        0 <= p[i,j] <= 1
        sum (1 <= j <= N) p[i,j] = 1

Thus the ‘transition’ of the system from Si to Sj has
probability p[i,j].

If a state sequence starts from Si it may or may not
return to Si; such a return is called a ‘recurrence’.
Let f[i,t] be the probability that the sequence returns
to Si for the FIRST time after the t’th transition (so
f[i,1] == p[i,i]).  Here t => 1 is thought of as ‘time’.
Then

        f[i] = sum ( 1 <= t ) f[i,t]

is the probability that the system returns to Si at
some future time (the probability that Si ever recurs).

A state Si is said to be persistent if f[i] == 1, and to
be transient if f[i] < 1.

A state Si is said to be no-return if f[i] == 0.

A state Si is said to be periodic if it is NOT no-return
and there is an integer s > 1 such that f[i,t] == 0 if
t mod s != 0.  The largest such s is the period of Si.

A state Si that is NEITHER no-return or periodic is said
to be aperiodic.

You have been asked to compute f[i] for the states Si of
a Markov Chain and determine if Si is transient or per-
sistent and if Si is no-return, periodic, or aperiodic.
In the periodic case you are to determine the period.


Input
-----

For each of several test cases, first a line containing
just the test case name, then a line containing just N,
and then N lines containing the probabilities in the
layout

        p[1,1] p[1,2] ... p[1,N]
        p[2,1] p[2,2] ... p[2,N]
        . . . . . . . . . . . .
        p[N,1] p[N,2] ... p[N,N]

1 <= N <= 100,  0 <= p[i,j] <= 1, and for each i,
sum p[i,j] over all j = 1.

Input probabilities may have many decimal places and
the lines containing them may be long.  Double precision
floating point will suffice for input and computation.

Input ends with an end of file.


Output
------

For each test case, the first a line containing an exact
copy of the test case name input line, then N lines each
with the format:

        f[#] = #.### X Y
```

```
where # denotes a digit, X is either ‘persistent’ or
‘transient’, Y is either ‘no-return’, ‘period #’ or
‘aperiodic’.

The only whitespace in the output are 4 single spaces,
2 surrounding the =, 1 before X, and 1 before Y.  The
N lines are in order of increasing f[#] index, i.e.,
f[1], f[2], ..., f[N].

The input will be such that a state i will be persistent
if and only if f[i] >= 0.9995, and there will be no
ambiguous cases.

WARNING: f[i] < 0.0005 does NOT mean the state is
no-return.  You must use a calculation not involving
f[i] to determine whether a state is no-return (use
your period calculation).


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

-- SAMPLE 1 --
2
0 1
1 0
-- SAMPLE 2 --
2
0.5 0.5
0 1
-- SAMPLE 3 --
2
0 1
0 1

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

-- SAMPLE 1 --
f[1] = 1.000 persistent period 2
f[2] = 1.000 persistent period 2
-- SAMPLE 2 --
f[1] = 0.500 transient aperiodic
f[2] = 1.000 persistent aperiodic
-- SAMPLE 3 --
f[1] = 0.000 transient no-return
f[2] = 1.000 persistent aperiodic


Note
----

For finite markov chains, persistent aperiodic states
are called ‘ergodic’.  For infinite markov chains,
persistent aperiodic states with finite expected time
of return are called ‘ergodic’, but there are also
persistent aperiodic states with infinite expected time
of return which called ‘null states’.
```

File: markov.txt
Author: Bob Walton <walton@seas.harvard.edu>
Date: Wed Oct 15 07:30:03 EDT 2014