```
Problems Index              Sat Oct 23 00:35:52 EDT 2004


Easy Problems:
--------------

    problems/passwords
        Find the right password for you.

    problems/tile
        Put those sub-windows in.

Medium Difficulty Problems:
---------------------------

    problems/drunkard
        Walk the walk and compute.

    problems/dicequiz
        Can you rotate with precision?

    problems/nearring2d
        Get your dog home.

Difficult Problems:
-------------------

    problems/opttile
        Put those sub-windows in optimally.

    problems/transducer
        Odds on words.
```

```
Password Pat
-------- ---

Password Pat is known for making slick passwords such
as
                @hwl2rt&tk

which is derived from the sentence

  At Hogwarts we love to roast tyrants and tea kettles.

by applying the following rules while reading the
sentence:

(1) On reading a non-special word, output its first
    letter in lower case.

(2) The special words and the single character to
    output are:

        and     &               zero    0
        or      |               one     1
        not     !               two     2
        equal   =               three   3
        plus    +               four    4
        minus   -               five    5
        times   *               six     6
        slash   /               seven   7
        dollar  $               eight   8
        percent %               nine    9
        at      @               to      2
                                for     4
                                ate     8

    Note these words are recognized even if some of
    their letters are capitals.

(3) On reading space characters, output nothing.

(4) On reading punctuation, output the punctuation,
    except on reading a period output nothing.

(5) Numbers are not permitted in the sentence (unless
    spelled out as words).

Pat does not limit herself to a single input sentence.
For example, the input

        fairly!         squarely!       I won?

produces the password:          f!s!iw?

You are to write a program that will apply Pat’s rules
to sentences to derive a password.

Input
-----

Lines each of which contains one or more sentences.
Words on the line are sequences of consecutive letters.
All input characters are letters, spaces, or one of the
punctuation characters .!?, .  No line is longer than
80 characters.  Input ends with an end of file.


Output
------

One line containing a password for each input line.  The
password on a line must be that derived by applying
Pat’s rules to the sentences in the corresponding input
line.  There are no spaces or tabs in any output line.
```

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

At Hogwarts we love to roast tyrants and tea kettles.
        fairly!         squarely!       I won?
Slash and burn politics is for the minus birds.
I want to replace foobar with fee, fie, foe, fum!


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

@hwl2rt&tk
f!s!iw?
/&bpi4t-b
iw2rfwf,f,f,f!
```

File: passwords.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Thu Oct 21 05:17:36 EDT 2004

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

```
    $Author: walton $
    $Date: 2004/10/21 09:30:47 $
    $RCSfile: passwords.txt,v $
    $Revision: 1.8 $
```

Tiling Problem

If sub-windows of a computer screen window are not supposed to overlap, determining the placement of these sub-windows can be difficult. This problem addresses a simple case of non-overlapping sub-window placement.

We will call the sub-windows ‘tiles’, and abstract the problem by considering windows and tiles to be squares of characters. Thus saying that a window (or tile) has size N means the window (or tile) consists of NxN characters.

The problem is, given a window of size N, and tiles named A, B, C, ... of sizes sA, sB, sC, ..., place the tiles in the window. The position of a tile is its upper left corner. The window is blank before any tile is placed, meaning that all its characters are the space character. When a tile is placed, its name, which is a single character, is copied into all the window characters occupied by the tile.

In this problem tiles are placed in order of their name, and a strict left-to-right top-to-bottom scan is used to find positions for tiles. The first tile, which is always tile A, is always placed in the upper left corner of the window. Then the scan proceeds from the position of the last tile placed until the first position is arrived at where the next tile can be placed, without overlapping any previously placed tile. That position is used as the position of the next tile. Each tile must be completely inside the window. If a tile cannot be placed by the scan, the tile is ignored, and not placed at all. The scan always resumes from the position of the last tile placed (except when placing the first tile), and the scan never goes up, and never goes to the left except just after going down.

Input

For each case, one or more lines containing non-negative integers in the following order:

```
    the size N of the window, 0 < N <= 80
    the sizes sA, sB, sC, ... of the tiles in order
    the value 0 (which ends the case description)
```

Each tile size s is such that 0 < s <= N. There may be at most 26 tiles, named A through Z, and their sizes are given in the order of their names. Numbers may be separated, preceded, and followed by any combination of spaces and tabs. A case may be spread across several lines. Input ends with an end of file.

Output

For each case, a line containing a single ‘-’ and nothing else, followed by the N lines of the window. Each window line consists of the character ‘|’ followed by the N characters of the window line followed by ‘|’, and nothing else. There are no spaces or tabs in the output, except for spaces in the window.

Sample Input

```
8 1 2 3 4 5 0
        8       5 4 3 2 1
        0
```

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

-
|ABBCCC  |
| BBCCC  |
|   CCC  |
|DDDD    |
|DDDD    |
|DDDD    |
|DDDD    |
|        |
-
|AAAAACCC|
|AAAAACCC|
|AAAAACCC|
|AAAAADDE|
|AAAAADD |
|        |
|        |
|        |
```

File: tile.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Thu Oct 21 05:38:49 EDT 2004

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

```
    $Author: walton $
    $Date: 2004/10/21 09:42:10 $
    $RCSfile: tile.txt,v $
    $Revision: 1.4 $
```

The 1D Drunkard
--- -- --------

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

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

The following is a classic pseudo-random number generator:

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

where

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

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

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

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

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

Note

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

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

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

```
Input
-----

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

The             W m seed

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

The             H m n seed

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

Input ends with an end of file.


Output
------

The first thing each command outputs is a line contain-
ing exactly one '-' and nothing else.  This separates
the command output from the previous output.

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

The histogram output by the 'H' command consists of one
line for p = -m, -m+2, -m+4, ..., m-4, m-2, m.  This
line contains

                p           H(p)           P(p)

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

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

            N(p,m) =   exp ( - p**2 / ( 2 * m ) )
                     / sqrt ( 2 * PI * m )

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

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

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


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

W 20 7456353
H 10 1000 276089259
```

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

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

```
File:       drunkard.txt
Author:     Bob Walton <walton@deas.harvard.edu>
Date:       Fri Oct 29 06:22:50 EDT 2004

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: hc3-judge $
    $Date: 2004/10/29 10:23:16 $
    $RCSfile: drunkard.txt,v $
    $Revision: 1.6 $
```

```
Dice Quiz
---- ----

You have become involved in writing software for a game
that is played with 6-sided dice.  A die is placed on
a board so that one of its faces is North, and the die
is then moved by rolling it to the North, East, South,
or West, so different faces are then on the top, bottom,
and sides.

A data base is needed to answer queries such as

        T1 N6 E?

which means, if 1 is on Top and 6 is to the North, what
digit is to the East?  Note that the order in which the
first two items are written does not matter, and

        N6 T1 E?

is the same query.

To make matters dicier, the die involved are non-stan-
dard.  They are described by lines such as

        D T1 B9 N4 E8 W3 S2

which says that in one of its positions, the die has
1 on top, 9 on the bottom, 4 to the North, 8 to the
East, 3 to the West, and 2 to the South.  Note that
the order of items (except for the D) does not matter,
so

        D E8 N4 W3 B9 S2 T1

describes the same die.  Also, a die has 24 possible
positions, and can be described in any one of these.


Input
-----

Lines each of which either describes a die or is a
query.  Each query is to be answered for the last die
described (the first line describes a die).  The faces
of the die can only have single digits, 0 through 9.
Two items in a line are separated by a single space,
and there are no spaces or tabs before the first item
or after the last.  Input ends with an end of file.


Output
------

The output is a exact copy of the input with each query
?  replaced with the face digit that is the answer to
the query.  In making the copy you can assume that the
‘?’ in each input query is the last character of the
query line.  You can also assume each query describes
a possible position of the current die.
```

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

D T1 B9 N4 E8 W3 S2
T1 N3 E?
N3 T1 E?
D E8 N4 W3 B9 S2 T1
T1 N3 E?
N3 T1 E?
N3 T1 W?
N3 T1 S?
N3 T1 B?


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

D T1 B9 N4 E8 W3 S2
T1 N3 E4
N3 T1 E4
D E8 N4 W3 B9 S2 T1
T1 N3 E4
N3 T1 E4
N3 T1 W2
N3 T1 S8
N3 T1 B9
```

File: dicequiz.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Wed Oct 20 10:21:47 EDT 2004

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

    $Author: walton $
    $Date: 2004/10/20 14:24:31 $
    $RCSfile: dicequiz.txt,v $
    $Revision: 1.7 $

Drop by When You are Near the Ring

Oliver lives on a ring, a rather large spinning circle of metal in space. He and his fellow Dogplovians get around in one-time-spaceships, which are discarded after a single use. Typically, to get home Oliver aims his current ship more or less at the ring, fires the motor till its empty, then waits till he is closest to the ring, hops in his personal spacepod, and motors straight home in it.

Oliver has lost his computer programs and desperately needs you to write him one that will calculate when he is closest to his ring. In Dogplovian coordinates, the ring is in the xy plain centered on the origin. The input is the point where Oliver fired his motor and the velocity achieved (instantaneously for computational purposes). For convenience assume the motor is fired at time 0.

Note that distance to the ring is measured from the spaceship to the nearest point on the ring, as once Oliver gets to the ring he takes the ‘circle train’ to his domicile. Note also that all distances are in dogbounds, and times in dogbarks, but you do not really need to use this knowledge.

Ah, and we almost forgot to mention. Luckily for you, Oliver lives in two dimensional space, and not three dimensional space.

Input

For each case, a single line containing the 5 numbers

```
        r x y vx vy
```

where r is the radius of the ring, (x,y) the point where the motor is fired at time 0, and (vx,vy) the velocity achieved at time 0. The velocity is constant after time 0.

An end of file terminates the input.

Output

For each case, a single line containing the 2 numbers

```
        t d
```

where t is the time Oliver’s spaceship is closest to the ring and d is the distance between the spaceship and the ring at that time. Both numbers must be printed with exactly 3 decimal places.

The input will be such that t > 0 is always true; i.e., the spaceship will never be headed away from the ring.

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

1.0 -1.00 2.00 1 0
1.0 0 2 0.5 -0.5
10.0 1 1 1 1
10.0 1 1 -0.1 -0.1

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

1.000 1.000
2.000 0.414
6.071 0.000
80.711 0.000


File:      nearring.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Thu Oct 21 07:23:18 EDT 2004

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: 2004/10/21 11:24:04 $
    $RCSfile: nearring2d.txt,v $
    $Revision: 1.5 $
```

```
Optimum Tiling Problem
------- ------ -------

This problem is a harder variant of the Tiling Problem
(short name ‘tile’).  You must read that problem before
reading this problem.  It is intended that you will
code the ‘tile’ problem before you code this problem,
and use the ‘tile’ code in the solution to this problem,
but there is no requirement that you do this.

By a placement of tiles we mean an order in which the
algorithm of the ‘tile’ problem tries to place the
tiles.  A placement can be labeled by giving the names
of the tiles in the order of the placement.  The ‘tile’
problem only tries one placement, the placement ABC...
in which the tiles are tried in order of their names.

In this problem you are asked to find a placement that
‘works’, in the sense that all tiles can actually be
placed, and none are ignored.

There may be more than one such placement.  For example,
if there are just two tiles and the placement AB works,
then so will the placement BA.  You are asked to find
the unique working placement that is first in lexical
(dictionary) order.  Thus you would find AB and not BA.


Input
-----

Same as the ‘tile’ problem.

This is a search problem.  The input is chosen so the
search will always succeed, and never fail, within the
contest problem time limit, provided you do some very
simple search tree pruning.  If your program does not
handle the sample input below very fast, you have not
pruned properly.


Output
------

For each case, a line containing nothing but the
placement order you found for the case.


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

10 3 3 2 4 4 6 2 0
10 4 3 4 6 3 0
26 2 2 2 2 2 2 2 2 2 2 2 2 2
   2 2 2 2 2 2 2 2 2 2 2 2 24 0

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

ABCGDFE
ABEDC
ABCDEFGHIJKLMNZOPQRSTUVWXY
```

```
File:       opttile.txt
Author:     Bob Walton <walton@deas.harvard.edu>
Date:       Thu Oct 21 06:10:04 EDT 2004

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: 2004/10/21 10:21:08 $
    $RCSfile: opttile.txt,v $
    $Revision: 1.4 $
```

```
Transducer Problem
---------- -------

An NDFT, or non-deterministic finite transducer, is an
NDFA, a non-deterministic finite automaton, with output.
We will first describe NDFA’s and introduce the notation
we will use, and then we will describe NDFT’s.  You
will be asked to simulate the execution of NDFT’s.

An NDFA consists of a labeled directed graph, with nodes
called ‘states’ and arrows called ‘transitions’, and two
designated nodes of the graph: the start and stop state.
We will use strictly positive integers, 1, 2, 3, ..., as
labels of states, and lower case letters, a, b, c, ...,
as labels of transitions.  We will denote a transition
as

        LABEL : ORIGIN -> TARGET

where LABEL is the transition label, ORIGIN is the label
of the transition origin state, and TARGET is the label
of the transition target state.  An NDFA can be describ-
ed by a sequence of such transition denotations and the
labels of the start and stop states.

A path through an NDFA is a sequence of transitions with
the target of each but the last being the origin of the
next transition in the sequence.  The origin of the path
is the origin of the first transition, and the target of
the path is the target of the last transition.  The
label of the path is the sequence of labels of the
transitions in the path.

Thus given the NDFA transitions:

        a : 1 -> 2
        b : 2 -> 3
        c : 3 -> 2
        c : 3 -> 4

we have the paths

        abc : 1 -> 2 -> 3 -> 2
        abc : 1 -> 2 -> 3 -> 4

A single state can be the origin of several transitions
with the same label.

An NDFA computes for each path label whether or not
there is a path from the start state of the NDFA to
the stop state of the NDFA.

An NDFT is an NDFA plus a value for each transition.  An
NDFT computes a value for each path, and computes a
value for a path label from all the paths with that
label between the start state and the stop state of the
NDFT.  In this problem all values will be floating point
numbers in the range from 0 to 1, which represent proba-
bilities.  Thus for us an NDFT assigns probabilities to
strings of transition labels.
```

```
We will use the notation

        LABEL : ORIGIN -> TARGET : VALUE

to denote a transition LABEL : ORIGIN -> TARGET with the
given VALUE.

The value of a path is the product of the values of the
transitions in the path.  The value of a path label is
the sum of the values of all paths with that label from
the start state to the stop state.

Thus given the NDFT:

        a : 1 -> 2 : 0.4
        a : 1 -> 4 : 0.6
        b : 2 -> 3 : 1.0
        b : 4 -> 5 : 1.0
        c : 3 -> 6 : 0.3
        c : 5 -> 6 : 1.0

        start state: 1
        stop state: 6

the following are the two possible paths from 1 to 6:

    abc : 1 -> 2 -> 3 -> 6 : 0.12    (= 0.4 * 1.0 * 0.3)
    abc : 1 -> 4 -> 5 -> 6 : 0.60    (= 0.6 * 1.0 * 1.0)

and the transducer computes the value 0.12+0.60 = 0.72
for the path label abc.
```

```
Input
-----

For each of several cases:

        a line containing:       N M START STOP
        N lines each denoting a transition
        M lines each containing a path label

where N, M, START, and STOP are integers greater than
zero.   Each case defines an NDFT with N transitions
and gives M path labels.  START and STOP are the
state labels of the start and stop states.  Input
ends with an end of file.

    1 <= N <= 1000
    1 <= M
    1 <= S <= 100 for any state label S
    transition labels are lower case letters
    path labels are 1 to 80 lower case letters
    0 <= VALUE <= 1.0 for any transition value


Output
------

For each case:

        a line containing nothing but a single '-'
        M lines each containing:  LABEL : VALUE

where the M lines correspond in order to the M input
lines containing path labels, LABEL is the path label
copied from the input line, and VALUE is the value
computed for that label by the NDFT.  Each VALUE
must have exactly 3 decimal places.
```

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

```
6 2 1 6
a : 1 -> 2 : 0.4
a : 1 -> 4 : 0.6
b : 2 -> 3 : 1.0
b : 4 -> 5 : 1.0
c : 3 -> 6 : 0.3
c : 5 -> 6 : 1.0
abc
abb
2 5 1 2
a : 1 -> 2 : 0.9
b : 2 -> 2 : 0.8
a
ab
abb
abbb
abbba
```

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

```
-
abc : 0.720
abb : 0.000
-
a : 0.900
ab : 0.720
abb : 0.576
abbb : 0.461
abbba : 0.000
```

Note

Our definitions of NDFA and NDFT are a more restrictive simplification of the standard definitions. The standard definitions allow empty labels on transitions (these do not appear in labels of paths containing the transition), and permit more than one stop state. Also the sum of the values of all transitions with the same origin and label is constrained to be equal to, or equal to or less than, 1. Lastly, values other than numbers may be used as long as multiplication, addition, 0, and 1 are defined (such a set of values is called a semi-ring). An example is sets of strings, where addition is set union, and multiplication of X and Y is the set of all strings made by concatenating a string from X and a string from Y. Addition must be commutative and associative, but multiplication must be merely associative. Multiplication must distribute over addition.

File: transducer.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Thu Oct 21 10:15:38 EDT 2004

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

$Author: walton $
$Date: 2004/10/21 14:15:51 $
$RCSfile: transducer.txt,v $
$Revision: 1.6 $