Problems Index Wed Oct 15 09:32:32 AM EDT 2008

The problems are in approximate order of difficulty, easiest first. The last two problems are particularly difficult.

problems/equity
So you think you know stocks?
Boston Preliminary 2008

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

problems/tableformatter
Simple work aids make life easier.
Boston Preliminary 2008

problems/buffalo
Lost in space? Call out the hound!
Boston Preliminary 2008

problems/paxos
Stable memory without disks.
Boston Preliminary 2008

problems/twolegs
A maze does the splits.
Boston Preliminary 2008

problems/supercounter
Tough counting.
Boston Preliminary 2008

problems/convexhull
A tight fit in 3D.
Boston Preliminary 2008

Equity

You have been asked to write a program that will print information that can be used to evaluate the financial situation of a company. The input is raw data about a company for a sequence of years, and the output is an income statement and balance sheet for the company for each year.

Input

The input is a set of lines each of which consists of a data type character followed by a datum. These are

N<company-name>
    All the data lines following, until the next N line or end of file, are for a company with the given name. All data for one company is grouped together; any two different N lines are for different companies. An N line is always followed by a Y line.
Y<current-year>
    All the data lines following, until the next N or Y line or end of file, are for the given 'current year'. The Y lines for one company are sequentially increasing; that is, every Y line but the first for a given company contains a value one greater than the value of the previous Y line for the company.
S<sales>
    The sales (i.e. revenue) of the company in the current year.
R<received>
    The amount of money received from sales for the given year. If you sell $1,000 worth and receive $900, then you are still owed $100, which is called a 'receivable'.
O<operating-cost>
    The cost of operating the company during the current year. This is the cost of manufacturing, administration, sales, advertising, etc. (but NOT the cost of capital expenditures, interest, dividends, or taxes).
P<paid>
    The amount of the operating cost you actually paid during the current year. If your cost was $1,000 and you paid $900, the you still owe $100, which is called a 'payable'. If instead you paid $1100, then presumably you had at least $100 of payables carried over from the previous year and paid off $100 of it.
C<capital-expenditures>
    Amount of money spent on capital (land purchases, construction, machinery, trucks, etc.) in the current year.
D<depreciation>
    The amount of capital that is to be charged against income in the current year. If you pay $30,000 for a truck in 2003, rather than charge it all against income in 2003, you may charge $10,000 in each of the three years 2003, 2004, and 2005 as 'depreciation'. If you pay $1,000 for land, however, you may never charge anything to depreciation, on the grounds that the land value does not depreciate.
I<interest>
    Interest paid on bonds in the current year.
T<taxes>
    Taxes paid in current year.
V<dividends>
    Dividends paid to stock holders in the current year.
E<new-equity>
    Amount taken in from venture capitalists or by selling new shares of common stock, minus amount paid out to venture capitalists or to buy back the company's shares of common stock, for the current year.

B<new-debt>
    Amount of bonds sold minus amount of bonds retired
    during the current year.
G<inventory-change>
    Value of goods produced for sale minus value of
    goods sold during the current year.

More than one company is described in the input. The N value is a character string (all the characters after the N until the end of line). All the other values are integers, which may be negative in some cases. All the integer values are in some unspecified unit, such as millions of dollars or thousands of dollars, except of course the years. If any of the dollar values are not given for a year, they are to be taken to be zero. E.g., in the first year of a company, we might give only Y, E, and C lines, and this means that all the other dollar values are zero.

Input ends with an end of file.

Output

The output is an income statement and a balance sheet for each company year for which data is provided in the input.

The format of the output is as given in the Sample Output. All numbers are integers (you can think of them as being in millions or thousands of dollars). All integers must be printed with their low order digit exactly in column 40. Only negative integers have a sign, and no high order zeros may be printed. Consecutive words and years in the output must be separated by a single space; and you may assume that this is true of company names given in N lines.

Each non-blank output line starts in either column 1 or column 3, and this indentation must be exactly as in the Sample Output. You can assume the N input line value is well formatted, and you should just copy that line with its initial 'N' removed to an output line.

There should never be two blank lines in a row, and blank lines must be included as indicated in the Sample Output. There MUST NOT be any blank line at the beginning or end of output (be careful of this).

Note that failure to follow any of the spacing rules will result in a 'Formatting Error' score for your program if everything else is correct.

In what follows, single capital letters such as S and I mean 'the S input line value' and 'the I input line value', etc. The output is computed as follows:

'Income Statement' for a given company and year:

    Operating Revenue: S
    Operating Cost: O
    Depreciation: D
    Operating Income: Operating Revenue - Operating Cost
                                        - Depreciation.
    Interest: I
    Taxes: T
    Earnings: Operating Income - Interest - Taxes.
    Dividends: V
    Transfer to Equity: Earnings - Dividends.

'Balance Sheet' for a given company and year:

    All these values are for the end of the current
    year. These values are all set to zero before the
    first year of a company (i.e, by each N input line).

```
    Assets:
        Cash: Cash from previous year + R - P - C + E
            + B - I - T - V.
        Inventory: Inventory from previous year + G.
        Receivables: Receivables from previous year
                    + S - R.
        Current Assets: Cash + Inventory + Receivables.
        Fixed Assets:  The Fixed Assets from the pre-
                       vious year + C - D.  (Essentially
                       the undepreciated capital).
        Total Assets: Current Assets + Fixed Assets.

    Liabilities:
        Payables: Payables from previous year + O - P.
        Current Liabilities: Payables
        Debt: Debt from previous year + B.
        Total Liabilities: Current Liabilities + Debt.

    Equity: Total Assets - Total Liabilities.

You may find it helpful to program the following check
into your program as a debugging aid:

    Equity = Equity from the previous year
           + Transfer to Equity + E + G

Note that neither assets nor liabilities may be
negative, and among input values only E, B, and G may be
negative, but the income, earnings, transfer to equity,
and equity values can be negative.  Mathematically it
would be possible for an asset like cash to go negative,
but it would be an accounting error if this happened.
```

Sample Input

```
------ -----

NBills’s Toothpaste Company
Y1950
E15
C20
B10
Y1951
S30
R27
O25
P24
D3
I1
G1
T1
V1
NGreat Flame Barbecue
Y1975
E5
C4
S10
R9
O6
P5
T1
G1
```

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

Bills’s Toothpaste Company

1950 Income Statement:
  Operating Revenue:                0
  Operating Cost:                   0
  Depreciation:                     0
  Operating Income:                 0
  Interest:                         0
  Taxes:                            0
  Earnings:                         0
  Dividends:                        0
  Transfer to Equity:               0

1950 Balance Sheet:
  Cash:                             5
  Inventory:                        0
  Receivables:                      0
  Current Assets:                   5
  Fixed Assets:                    20
  Total Assets:                    25
  Payables:                         0
  Current Liabilities:              0
  Debt:                            10
  Total Liabilities:               10
  Equity:                          15

1951 Income Statement:
  Operating Revenue:               30
  Operating Cost:                  25
  Depreciation:                     3
  Operating Income:                 2
  Interest:                         1
  Taxes:                            1
  Earnings:                         0
  Dividends:                        1
  Transfer to Equity:              -1

1951 Balance Sheet:
  Cash:                             5
  Inventory:                        1
  Receivables:                      3
  Current Assets:                   9
  Fixed Assets:                    17
  Total Assets:                    26
  Payables:                         1
  Current Liabilities:              1
  Debt:                            10
  Total Liabilities:               11
  Equity:                          15

Great Flame Barbecue

1975 Income Statement:
  Operating Revenue:               10
  Operating Cost:                   6
  Depreciation:                     0
  Operating Income:                 4
  Interest:                         0
  Taxes:                            1
  Earnings:                         3
  Dividends:                        0
  Transfer to Equity:               3

1975 Balance Sheet:
  Cash:                             4
  Inventory:                        1
  Receivables:                      1
  Current Assets:                   6
  Fixed Assets:                     4
  Total Assets:                    10
  Payables:                         1
  Current Liabilities:              1
  Debt:                             0
  Total Liabilities:                1
  Equity:                           9
```

Remarks:

The above comes mostly from 'The Interpretation of Financial Statements' by Benjamin Graham. However the above is an over-simplification, and there are also terminological problems and accounting method disputes that the author of this problem is not qualified to deal with.

To take one example, a company with a good balance sheet should have perhaps twice as much Current Assets, which might be interpreted as assets that could be converted into cash in a year, than Current Liabilities, which might be interpreted as liabilities that must be paid in a year. Above we assumed that all bonds were long term and none were current liabilities, but some modern companies have taken to issuing lots of short term (e.g., three month) bonds which should be listed as Current Liabilities.

Even a real balance sheet is an over-simplification. For example, some inventory may be unsellable and have to be 'written off' as a loss eventually, and similarly some receivables may be uncollectible. The networking hardware companies leased much of their hardware to the dot-com bubble companies and had to write off large amounts of receivables when the dot-com bubble burst.

File: equity.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Wed Oct 15 09:11:05 EDT 2008

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

```
$Author: walton $
$Date: 2008/10/15 13:18:19 $
$RCSfile: equity.txt,v $
$Revision: 1.7 $
```

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

$$x(i+1) = A*x(i)^2 + B*x(i) + C \quad (\text{modulo } n)$$

$$x(0) = D$$

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@deas.harvard.edu>
Date:      Wed Oct 15 09:18:40 EDT 2008

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: 2008/10/15 13:20:51 $
    $RCSfile: birthday.txt,v $
    $Revision: 1.7 $
```

Making a Table Formatter
------ - ----- ---------

Olivia went to work for State Report Writers Inc. and found herself writing a lot of plain ASCII text files containing tables that looked like:

```
+-------------+--------+--------+--------+
| Year        |   2001 |   2002 |   2003 |
+-------------+--------+--------+--------+
| Population  | 37,452 | 37,459 | 37,620 |
+-------------+--------+--------+--------+
| Number of   | 13,645 | 13,652 | 13,684 |
| Households  |        |        |        |
+-------------+--------+--------+--------+
```

Getting things all lined up was becoming a real pain, so she wants a program to do it for her.

The program takes as input a text file whose tables are defective, and outputs the file with the defects corrected. The permitted defects are:

(1) Lines beginning with + can have any garbage after the +, and will be converted into proper row separating lines.

(2) If the first character of a table line is not + or |, it will be assumed that an initial | is missing from the line.

(5) The |'s and the text in a line need not be properly aligned, except a | that should not appear at the very beginning of a line may not be moved to the beginning of the line.

(4) A trailing sequence of |'s and spaces may be missing from a line.

(5) There may be empty columns in the table (columns all of whose entries are blank). These empty columns should be deleted.

(6) The last + line of the table may be omitted.

The program reads its input, outputs table lines after reformatting them, adds a table end + line if necessary, and outputs exact copies of any non-table lines. A table begins with a line whose first character is +, and ends just before the next blank line or file end.

Text in a table box should be separated by at least one space from any |. Within each line, the text in a box should be left adjusted if it contains any letters, and right adjusted otherwise. Note that text in a box may be left adjusted on one line and right adjusted on another line; each line is treated independently as far as text adjustment is concerned. Any spaces embedded inside the text (i.e., not next to |'s or line ends) should be preserved. All table lines should have the same length, and this should be the minimum length consistent with the given rules.

To simplify coding the only whitespace character allowed in a line is the single space character, and lines are not permitted to end with space characters. Thus a blank line will be completely empty. This rule applies to input lines, and you must apply it to output lines.

If you get only the spacing wrong, your program will be given the 'Formatting Error' score.

Input

The text to be reformatted. This text can contain any number of tables. No input line will be longer than 80 characters. The text ends with an end of file. The input will be such that every table contains at least one non-empty column. The input ends with an end of file.

Output

The input text after it is reformatted. Note that output lines may be longer than 80 columns due to reformatting, but the input will be such that no output line is longer than 132 characters.

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

```
FLEAS IN THE MIDDLE COUNTIES

by Dr. Troubledstat
edited by Olivia Oliviana

+
County | 2001 | 2002 | 2003
+
Upper Middle | 4,589,290 | 2,976,384 | 10,000,000
+
Middle Middle | 7,671,004 | 5,804,027 | 20,000,000
+
Lower Middle | 8,612,920 | 5,790,468 | 1,764,893
+

This is a

          TEST TABLE

+98435632-6987234899-8g544285-9245724||--
|0123456789      ||0123456789
123 || $789.87 |
A123
++++++++++++++++++++
|     H21 ||       x    y   z |||
foo
|||*&^%$#@
```

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

FLEAS IN THE MIDDLE COUNTIES

by Dr. Troubledstat
edited by Olivia Oliviana

County	2001	2002	2003
Upper Middle	4,589,290	2,976,384	10,000,000
Middle Middle	7,671,004	5,804,027	20,000,000
Lower Middle	8,612,920	5,790,468	1,764,893

This is a

TEST TABLE

0123456789 123 A123	0123456789 $789.87
H21 foo	x y z *&^%$#@

```
File:      tableformatter.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Wed Oct 15 09:21:16 EDT 2008

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: 2008/10/15 13:25:02 $
    $RCSfile: tableformatter.txt,v $
    $Revision: 1.9 $
```

Buffalo Finder
------- ------

Ranchy Flatman is a buffalo rancher who has lived on the Plains for decades. He has a ranch on which he keeps buffalo in a field with three sides. Each side is bounded by a perfectly straight fence which actually extends in both directions well beyond Ranchy's field, and which serves as a boundary between various fields owned by various other people. In all there are 7 fields, Ranchy's triangular field, and 6 neighboring fields.

Sometimes Ranchy loses track of a buffalo. Then he sends his trusty Beagle Issy ('I Smell You') out to find the errant buffalo. Issy wears a GPS receiver and a radio, and this sends Issy's position back to Ranchy. When Issy finds the buffalo, she stops, and Ranchy then knows the buffalo's GPS coordinates. If the buffalo has gotten lost in Ranchy's field, Ranchy goes out to find the buffalo, but if the buffalo is in a neighbor's field, Ranchy must call up the neighbor who will go with Ranchy to retrieve the Buffalo and Issy.

In order to make this work, Ranchy's daughter has worked out the following naming system for fields, and programmed the family computer to tell Ranchy which field Issy is in. The corners of Ranchy's triangular field are given the names 1, 2, and 3 in clockwise order, and the fences are given names 12, 23, and 31 in clockwise order. A given field can be either to the left or right of a given fence. So we can give a field a name of the form

D12 D23 D31

where Dxy is 'L' if the field is to the left of fence xy and 'R' if the field is to the right of fence xy when traveling in the direction from x to y. Thus Ranchy's triangular field is named RRR and if you cross fence 12 from this field you enter field LRR.

To find out which field Issy is in, the GPS coordinates of Issy and the corners 1, 2, and 3 are used. The GPS coordinates are treated as integer coordinates of points in a flat plane.

Due to an unfortunate accident, Ranchy's daughter's computer program has been lost, and as she is off at college and in the middle of exams, you have been tasked to replace it.

Input

For each test case, one line of the form

x1 y1 x2 y2 x3 y3 xi yi

where (x1,y1), (x2,y2), (x3,y3) are the coordinates of the corners 1, 2, and 3, respectively, and (xi,yi) are Issy's coordinates. All coordinates are integers. Input ends with an end of file.

For simplicity, the input will be such that Issy is never exactly on a fence.

Output

For each test case one line containing just the name of the field containing Issy.

```
Sample Input
------ -----
-3 -3 0 6 3 -3 0 0
-3 -3 0 6 3 -3 -5 0
-3 -3 0 6 3 -3 0 10
-3 -3 0 6 3 -3 5 0
-3 -3 0 6 3 -3 10 -4
-3 -3 0 6 3 -3 0 -4
-3 -3 0 6 3 -3 -10 -4


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

RRR
LRR
LLR
RLR
RLL
RRL
LRL


File:      buffalo.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Wed Oct 15 03:05:19 EDT 2008

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: 2008/10/15 07:06:00 $
    $RCSfile: buffalo.txt,v $
    $Revision: 1.6 $
```

The PAXOS Algorithm
--- ----- ---------

The PAXOS algorithm was developed by Leslie Lamport (and independently by others) to make a decision rapidly in a distributed computer system where any processor can fail. You have been asked to simulate the algorithm in order to get a sense of how it works.

A distributed system is a set of processes connected to each other by communication channels. For the purposes of this problem, we assume there are two channels between every pair of processes, one channel going in each direction.

In order for a distributed system to run as fast as possible, it should be 'asynchronous'. This means that as soon as a process receives a message, it computes its new state and sends appropriate messages to other processes, without waiting for any particular time. However, there is a theorem that any asynchronous distributed algorithm will stop dead if just one process fails at exactly the wrong moment. Thus asynchronous distributed algorithms can fail if just one of their processes fails.

The PAXOS algorithm can be thought of as an asynchronous decision making algorithm that can be run more than once to make the same decision. That is, there can be more than one instance of the algorithm. If one instance appears to be failing, another instance can be started. The algorithm has the critically important property that if several of the instances succeed in coming to a decision, they will all come to the SAME decision.

The algorithm description is as follows:

(1) For our purposes, the decision is to be made between two values, labeled 'B' and 'C'. In the real world the decision is most often between 'aborting' and 'committing' a data base transaction, so you can think of 'B' as meaning 'abort' and 'C' as meaning 'commit'.

(2) Each algorithm instance has a single master process. This process will make the decision and send it to all the the other processes. The master of the first instance is generally chosen by the nature of the decision being made, and the masters of later instances are simply processes that, using clocks, have decided that previous algorithm instances are unduly delayed and a new algorithm instance needs to be started.

    All the processes other than the master for an instance are called the slaves of that instance. A process can be the master of some algorithm instances and a slave in other instances.

(3) Each instance is assigned an identifier. No two instances may have the same identifier, and the identifiers are ordered so that instances started later are later in the ordering. In the real world identifiers are typically numbers whose high order bits are the time of day and whose low order bits are a unique process identifier.

(4) Processes communicate by messages sent over channels. We assume there are n processes numbered from 1 through n, and there is exactly one directed channel connecting message sending process j to message receiving process k, for every pair of processes j and k with j != k. We assume each channel delivers messages reliably and in the order they were sent, BUT, with arbitrary delay. Such a channel can be built on top of unreliable communications by an appropriate channel protocol, which we do not consider here (in the real world the internet TCP protocol would likely be used).

We assume each channel can hold exactly one message, which has been sent but not received. We assume that if a second message is sent to the channel when the channel is not empty, any previous message in the channel is discarded and lost. This behavior just simplifies the code of our simulation; a real world channel probably would not discard messages very often.

(5) The algorithm messages have the following formats, where lower case letters are variables and upper case letters are constants:

m s N i
New-instance (N) message, sent from instance master process m to instance slave process s, announcing that a new instance has been created with identifier i.

s m A i pd pi
(New-instance) acknowledgment (A) message, sent from instance slave s to instance master m, for the instance with identifier i. pd and pi are the values of variables maintained by the slave as part of its state (see below).

m s P i d
Proposal (P) message, sent from instance master process m to instance slave process s, for the instance with identifier i, proposing that the instance decision be d (d == B or C).

s m Q i
Proposal acknowledgment (Q) message, sent merely to acknowledge the above proposal (P) message.

m s F i
Final (F) message, sent from instance master process m to instance slave process s, for the instance with identifier i, stating that the proposed decision for instance i has become the final decision of the set of all instances, and no subsequent instance will ever propose a different decision.

(6) Each process maintains the following variables:

imax — The latest (maximum) instance identifier for which the process has either received or sent a new-instance (N), proposed (P), or final (F) message. Initialized to -1. All instance identifiers are integers >= 0.

pd — The decision value in the last proposal (P) message the process has either sent or has received and not ignored. Initialized to X (meaning no proposal has been received).

pi — The instance identifier in the last proposal (P) message the process has either sent or has received and not ignored. Initialized to -1.

Note that a master maintains these variables as if it were also a slave that receives and does not ignore all the messages the master sends.

Here -1 is treated as an instance identifier value that is less than any actual instance identifier.

(7) Any received message is ignored (discarded) if it is a message whose instance identifier i is LESS THAN the receiver's current imax variable value.

(8) There are n processes. Let m be the smallest integer such that 2m > n. Any set of processes with m members is called a 'majority'.

PAXOS uses the fact that any two majorities must overlap.

(9) The instance algorithm is:

(a) The master picks an instance identifier i based on the current time. This must be greater than the imax variable value of the master, and should be greater than the identifier of any previous instance.

(b) The master sends new-instance (N) messages to all slaves. Upon receiving and not ignoring the N message, a slave sends an acknowledgment (A) message back to the master containing the slave's pd and pi variable values.

(c) As soon as the master receives m-1 acknowledgment (A) messages, it makes a proposed decision. It knows at this point the pd and pi values of m processes, the m-1 acknowledging slaves and the master itself. If all pd values are X, meaning 'not yet set', the master is free to make any decision it wants (this will be the case for the first instance). Otherwise the master chooses the pd value whose associated pi value is greatest.

(d) The master sends proposal (P) messages to all slaves containing the new proposed decision. Any slave receiving and not ignoring this message returns a proposal acknowledge (Q) message to the master. During this process activity the pd and pi variable values of the master and all acknowledging slaves are updated.

(e) As soon as the master receives m-1 proposal acknowledgment (Q) messages, it sends a final (F) message to all slaves. Note that in the our simulation slaves ignore final (F) messages, though in the real world they would not.

Event Specification

The simulation you have been asked to perform represents algorithm execution as a sequence of events. The events are numbered 1, 2, 3, etc., and the simulation input contains a list of event specification lines that are each one of the following:

N m d
Process m becomes the master of a new instance whose instance identifier is the number of the current event and whose decision will be d (B or C) if the master is free to choose in step (c) above. The event consists of the new master updating its variables and sending new-instance (N) messages to every other process.

R j k
Process k receives the message in the channel whose sending process is j and whose receiving process is k, if there is a message in that channel. If there is no message, there is no event. If there is a message, the event consists of the receiving process updating its variables and sending messages to other other processes as specified by the algorithm steps above.

Events in general send messages (except for events where a received message is ignored). Sending a message consists of placing the message in the appropriate channel. If necessary any previous message in the channel is first discarded.

Input

The input consists of several test cases. Each test case consists of

a line containing the name of the test case
a line containing the number n of processes
zero or more event specification lines
a line containing only the character 'E'

Here 2 <= n <= 32. The input terminates with an end of file.

Output

For each test case the line containing the name of the test case is output. Then for each event the output is as follows:

(a) If the event is the creation of a new instance, a line of the form 'e: NEW INSTANCE m d' is output, where e is the event number, and also the identifier of the new instance, and m and d are taken from the 'N m d' event specification.

(b) If the event is the reception of a message, a line is output of the form 'e: message action' where e is the event number (1, 2, 3, etc.), the 'message' is the received message in the message format given above with single spaces surrounding it and separating its parts, and the 'action' is one of the following words:

IGNORED	The message is ignored as per (7) above.
COMMITTING	The message is the m-1'st proposal (P) message to be received and not ignored by a slave for a given instance.
ACCEPTED	All other cases.

Note that if for an event specification 'R j k' the channel is empty, there is no event, there is no output, and the current event number is not incremented.

In general, items in an output line are to be separated by a single space character, except there is no space before a ':'. There should be no other whitespace characters in any output line. You may assume the test case name input line is property formatted, and you should just copy it to the output.

At the end of each test case print one empty line. This will appear before the name of the next test case, or before the end of the output file.

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

```
ERRORLESS CASE
3
N 1 C
R 1 2
R 1 2
R 1 3
R 2 1
R 3 1
R 1 2
R 1 3
R 2 1
R 3 1
R 1 2
R 1 3
E
ONE FAILURE CASE
3
N 1 C
R 1 2
R 1 3
N 2 B
R 2 1
R 3 1
R 2 3
R 1 2
R 3 2
R 2 1
R 2 3
R 1 2
R 3 2
R 2 1
R 2 3
E
```

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

```
ERRORLESS CASE
1: NEW INSTANCE 1 C
2: 1 2 N 1 ACCEPTED
3: 1 3 N 1 ACCEPTED
4: 2 1 A 1 X -1 ACCEPTED
5: 3 1 A 1 X -1 ACCEPTED
6: 1 2 P 1 C COMMITTING
7: 1 3 P 1 C ACCEPTED
8: 2 1 Q 1 ACCEPTED
9: 3 1 Q 1 ACCEPTED
10: 1 2 F 1 ACCEPTED
11: 1 3 F 1 ACCEPTED

ONE FAILURE CASE
1: NEW INSTANCE 1 C
2: 1 2 N 1 ACCEPTED
3: 1 3 N 1 ACCEPTED
4: NEW INSTANCE 2 B
5: 2 1 N 4 ACCEPTED
6: 3 1 A 1 X -1 IGNORED
7: 2 3 N 4 ACCEPTED
8: 1 2 A 4 X -1 ACCEPTED
9: 3 2 A 4 X -1 ACCEPTED
10: 2 1 P 4 B COMMITTING
11: 2 3 P 4 B ACCEPTED
12: 1 2 Q 4 ACCEPTED
13: 3 2 Q 4 ACCEPTED
14: 2 1 F 4 ACCEPTED
15: 2 3 F 4 ACCEPTED

[The last output line is blank.]
```

Remarks

A non-distributed reliable system can be made from a computer and a disk. The computer runs instances of an algorithm which makes decisions. To make a decision, an instance first proposes it, then writes it to disk, then declares the decision final. If the computer crashes, a new instance of the algorithm is started, which begins by reading the disk to find out all the previously proposed decisions. Since it does not know the extent to which these have been acted on, it must assume each of these proposed decisions is final.

The disk is referred to as ‘stable storage’, because it survives crashes and provides reliability.

The PAXOS algorithm uses the set of processes to implement stable storage (without any disks). The proposal (P) messages write the proposed decision to stable storage (the set of processes), and the proposal acknowledgment (Q) messages confirm that the decision has been written. In the non-distributed case the proposed decision has been successfully written to stable storage when its last bit has been successfully copied to disk. In the PAXOS case the proposed decision has been successfully written to stable storage when the m-1’st slave which will not ignore the proposal’s P message has received the P message. At this point any subsequent instance will read the proposed decision. So at this point the proposed decision is ‘committed’. The master knows it has successfully written stable storage when it receives the m-1’st Q message.

The new instance (N) messages and their acknowledgment (A) messages correspond to reading stable storage. The master knows it has successfully read stable storage when it receives the m-1'st A message, and any proposed decision read is that associated with the most recent instance known to any of the slaves that sent the A messages. Because two majority sets of processes must overlap, any committed proposed decision becomes known to the master. Here we are using the fact that if a committed proposed decision is overwritten by a new proposed decision, the new proposed decision must be the SAME as the old proposed decision, by step (c) of the algorithm.

One can formalize all this in an inductive mathematical proof that using PAXOS, once a proposed decision is committed by an instance no future instance can propose any other decision.

The analogy we have drawn between the non-distributed and distributed cases is not precise. For example, in the distributed case several instances can run simultaneously (in DIFFERENT processes), and a proposed decision that is never committed can end up being read as the latest proposed decision by a subsequent algorithm instance.

File: paxos.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Wed Oct 15 09:25:31 EDT 2008

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

$Author: walton $
$Date: 2008/10/15 13:26:34 $
$RCSfile: paxos.txt,v $
$Revision: 1.10 $

The Two Legged Maze

A two legged maze is a board of NxN squares, with each square labeled by a single lower case letter. The problem is to go by a path from a start square to a goal square. The path is a sequence of horizontal and vertical moves to adjacent squares. The path is divided into two parts, the 'first leg' followed by the 'second leg'. Each leg is a sequence of moves. In the first leg all moves must be from a square labeled X to a square labeled Y where Y follows X in the alphabet, or in short, X<Y. In the second leg all moves must be from X to Y where Y precedes X in the alphabet, or X>Y. Either leg can have zero moves.

The board columns are numbered 1, 2, ..., N and the board rows are numbered 1, 2, ..., N. A square has coordinates (r,c) where r is the square's row number and c is its column number, (1,1) is the upper left corner, and (N,N) is the lower right corner. The start square is (sr,sc) and the goal square is (gr,gc).

Diagonal moves are NOT allowed.

Input

For each of several test cases, a single line containing the 5 numbers

```
N sr sc gr gc
```

where 1 <= N <= 50, followed by N lines each containing just N lower case letters. These lines are the rows of the board, with row 1 first and row N last. The N lower case letters on a line are the labels of the squares in the line's row, with the first letter in the line being for column 1 and the last being for column N.

An end of file terminates the input.

Output

For each case, a single line containing nothing but the minimum number of moves required by any path from the start to the goal. Or if there is no such path, then the line contains just the word 'impossible'.

Example Input

```
5 1 1 5 5
abcde
fghij
klmno
pqrst
uvwxy
5 1 2 4 5
abken
ywxyz
abekw
yxieb
pqude
5 4 2 5 5
abkzn
ywxyx
abekw
ymief
pqude
```

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

8
6
impossible


File:      twolegs.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Wed Oct 15 03:08:09 EDT 2008

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: 2008/10/15 07:08:22 $
    $RCSfile: twolegs.txt,v $
    $Revision: 1.4 $
```

Superstring Counter
---------- -------

A superstring of a set of strings is a string containing every member of the set (possibly with overlapping) as a substring. You have been asked to find the number of superstrings with a given length of a given set of substrings. All strings consist of lower case letters.

Input

For each test case, one line containing just ‘n m’, where n is the length of the superstring and m is the number of substrings to be included in it, followed by m lines, each containing nothing but one substring. 1 <= m <= 10, 1 <= n <= 40. Substrings contain only lower case letters. The input ends with an end of file.

Output

For each test case, one line containing the number of superstrings containing just n lower case letters. The input will be such that the output will be less than 2^31.

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

```
4 2
ab
cd
3 1
aa
```

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

```
2
51
```

File: supercounter.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Wed Oct 15 09:27:16 EDT 2008

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

```
$Author: walton $
$Date: 2008/10/15 13:27:26 $
$RCSfile: supercounter.txt,v $
$Revision: 1.5 $
```

3D Convex Hull
-- ------ ----

You are given a large number of points in 3-space and must find the convex hull of these points.

Input

For each test case, one line containing the name of the test case, followed by a line containing just ‘n’, the number of points you are given, followed by n lines each containing ‘x y z’, the coordinates of one of the points. The points are numbered 1 through n in the order that they appear in the input file.

The xyz coordinates are all integers in the range from -1000 through +1000. 4 <= n <= 10,000.

The input ends with an end of file.

Output

For each test case, one line containing just the name of the test case, followed by a line containing just ‘m’, the number of edges in the convex hull, followed by m lines, each containing ‘i j’, where i and j are the numbers of the points that are the vertices of one of these edges. The input will be such that 4 <= m <= 100. An edge, by definition, is the edge of a face of the convex hull, and such a face can be any convex polygon.

To simplify the code, the input will be such that each face is a triangle and all points on the convex hull will be vertices of faces. Thus no point will be in the interior of a face or on the line between two other points that are on the hull.

All the hull face edges must be output in SORTED order. For an edge output as ‘i j’, i < j is required. Edges with lower i values must be output first, and among edges with the same i values, those with lower j values must be output first.

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

```
TETRAHEDRON IN A CUBE
4
0 0 0
1 1 0
0 1 1
1 0 1
OCTAHEDRON
9
0 0 0
2 0 0
0 2 0
2 2 0
1 1 -2
1 1 2
1 1 -1
1 1 0
1 1 1
```

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

TETRAHEDRON IN A CUBE
6
1 2
1 3
1 4
2 3
2 4
3 4
OCTAHEDRON
12
1 2
1 3
1 5
1 6
2 4
2 5
2 6
3 4
3 5
3 6
4 5
4 6
```

```
File:      convexhull.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Wed Oct 15 09:30:27 EDT 2008

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: 2008/10/15 13:31:45 $
    $RCSfile: convexhull.txt,v $
    $Revision: 1.7 $
```