Problem SP99-NZ1 Isotope Composition

An element is characterised chemically by its atomic number (the number of protons in the nucleus), however most elements occur in two or more *isotopes*— atoms with the same number of protons but differing numbers of neutrons. Isotopes are indistinguishable chemically, but become important in *mass spectrometry*. A mass spectrometer is a device which bombards a compound to produce a series of charged fragments and then sorts them according to their mass. Because of the presence of isotopes, fragments with different compositions may have the same mass and fragments with the same composition may have different masses. The proportions of the various naturally occurring isotopes of the common elements is known very accurately and this allows us to calculate the range and relative proportions of masses that a given fragment may exhibit.

The composition of a fragment is written as a chemical formula — a list of symbols representing entities followed by the number of times each entity occurs if it occurs more than once. Entities are typically elements in which case their symbols are either a single upper case letter (e.g. N for Nitrogen) or an upper case letter followed by a lower case letter (e.g. Na for Sodium), however frequently occurring groups of elements will also be given symbols, thus the ethyl grouping (C2H5) will be referred to as Et. Parentheses are used to delimit recurring groups of entities; a number ≥ 2 must follow every closing parenthesis to specify the number of times the group occurs. Thus valid formulae include H2O (water), H2SO4 (sulphuric acid), EtOH (ethanol, the 'alcohol' in drinks), (C2H5)2O (diethyl ether, the 'ether' used as an anaesthetic.

Write a program that will determine the range of weights and their percentage occurrence for a given formula.

Input

The input file will consist of two parts. The first part will define a list of symbols and either their isotopic composition (for elements) or their formula (for pseudo-elements). These formulae may be in terms of elements, other pseudo-elements or both. All symbols mentioned will be defined before use and there will not be any circularities. The symbol will occur in the first two character positions (the first character will be an upper case letter, the second will be a blank or a lower case letter) followed by a blank and either an '=' (for a pseudo-element) followed by a chemical formula or a ':' followed by the number of isotopes (in the range 1..5) followed in turn by that number of (atomic weight, abundance) pairs in order of increasing weight. Atomic weights are given as integers and abundances as percentages, i.e. as real numbers in the range (0.00, 100.00]. This part of the file is terminated by a line containing only ##.

The second part of the file contains a series of lines containing chemical formulae (one per line) according to the above rules. Lines will consist of no more than 60 characters, and no formula will consist of more than 60 atoms. This part of the file will also be terminated by a line consisting only of ##. This line should not be processed.

Output

For each formula (line) in the input, echo it to the output, followed by a list of the masses of all fragments that occur with an abundance greater than or equal to 0.001% together with their abundances, in order of increasing mass. The masses should be right justified in a field of width 5 and the abundances right justified in a field of total width 8 with 3 digits after the decimal point. Leave one blank line after each block of output corresponding to one formula. Follow the example shown below.

Sample Input

```
C  : 2 12 98.93 13 1.07
H  : 2 1 99.99 2 0.01
S  : 4 32 94.93 33 0.76 34 4.29 36 0.01
Me = CH3
Et = MeCH2
##
H2
C2H4
C6H14
Et
C2H5
C(CMe3)4
S
```

Sample Output

```
H2
     2  99.980
     3   0.020

C2H4
    28  97.832
    29   2.155
    30   0.012

C6H14
    86  93.618
    87   6.206
    88   0.173
    89   0.003

Et
    29  97.823
    30   2.165
    31   0.013

C2H5
    29  97.823
    30   2.165
    31   0.013

C(CMe3)4
   240  82.987
   241  15.557
   242   1.376
   243   0.076
   244   0.003

S
    32  94.939
    33   0.760
    34   4.290
    36   0.010
```

Problem SP99-NZ2 Resistance is Facile

An electronic workshop is going to have several boxes of resistors. They may have as few as four boxes or as many as twenty; they haven't decided yet. Each box will hold at least 2,500 resistors of a particular value, different for each box. One box will hold “1 thousand ohm” resistors. They haven't decided what values to use for the other boxes yet.

If you combine resistors in series, their values add up. For example, if you combine (in any order) three 500 thousand ohm resistors, two 200 thousand ohm resistors, a 50 thousand ohm resistor, two 20 thousand ohm resistors, a 5 thousand ohm resistor, and two 2 thousand ohm resistors, you get the equivalent of a 1999 thousand ohm resistor. Similarly you could replace the 5 thousand ohm resistor with two 2 thousand ohm resistors and one 1 thousand ohm resistor or one 2 thousand ohm resistor and three 1 thousand ohm resistors or five 1 thousand ohm resistors without changing the total resistance. In order to decide how many boxes to use and what to put in them, the workshop want to find out how easy it will be to make up various values, given a set of boxes. That is, given a number of boxes b, and resistor values $1=v_1 < v_2 < \ldots < v_b$ in thousands of ohms, they want to know how many different ways there are to make up an n thousand ohm resistance using those values in the boxes, for several values of n. Resistors of the same value cannot be told apart.

Input

Input consists of a series of integers, one per line. The first integer is b, which will be 4 to 20 inclusive. The next b lines contain an ascending sequence of numbers, starting with 1, representing v_1 to v_b, in that order. This list is followed by integers in the range 1 to 999 representing the values of the resistances we wish to make. The list is terminated by the integer 0 which should not be processed.

Output

For each value of n in the input output the number of ways to make up an n thousand ohm resistance, left justified on a line by itself.

Sample Input

```
9
1
2
5
10
20
50
100
200
500
1
5
10
42
137
999
0
```

Sample Output

```
1
4
11
271
451
15154
6295435
325581402
```

Problem SP99-NZ3 The Game of Tarot

Playing cards, used for trick-taking games, came to Europe from the Islamic world in the late 14th century. Islamic card packs had four suits: Cups (C), Coins (N), Swords (S), and Polo-Sticks. At that time Europeans hadn't the faintest idea what a polo stick was, so they called that suit Batons (B). Each suit consisted of ten numeral cards (A, 2-10), and three court cards: King, Knight, and Knave. European card players experimented with doubling the court cards, so that there was a King (K), Queen (Q), Knight (N), Dame (D), Page (P), and Maid (M). It was this form, with 16 cards per suit, for a total of 64 suit cards, which was used in the oldest known Tarot decks. Modern cartomantic packs drop the Maid and Dame, so cartomancers are quite literally not playing with a full deck. For ordinary card games, Europeans reverted to a 52 card pack, dropping the Knight, Dame, and Maid.

The trump Tarot cards were invented sometime in the first half of the 15th century. The order of the trumps has never quite stabilised, but the pictures were fairly stable for a while. The following table lists the names of the trumps, from the oldest sources, together with single letters chosen to represent them.

A	the Angel (of the last judgement)	M	the Moon
B	the Bagatelle, or the mounteBank	N	the devil (old Nick)
C	the (victor's) Chariot	O	the Popess (sic.)
D	the Empress	P	the Pope
E	the Emperor	R	the wheel (Rota) of fortune
F	Fortitude	S	the Sun
G	death (Grave)	T	Temperance
H	the Hanged man, or Traitor	U	hell (or fire)
I	the Fool, or Idiot	V	time
J	Justice	W	the World
L	Love	X	the star

Cards will be represented in the input by a face value (A2–9TMPDNQK for the suits and A–JL–PR–WX for the trumps) and a suit (T, C, N, S, or B). Thus TB is the ten of batons, and BT is the Bagatelle.

Ranking and Score.

Each card has a rank and a value. Rank determines which cards win, value determines how much they win. In Swords and Batons, the cards are ranked (in descending order): King, Queen, Knight, Dame, Page, Maid, 10, 9, 8, 7, 6, 5, 4, 3, 2, Ace. In Cups and Coins, the cards are ranked (in descending order): King, Queen, Knight, Dame, Page, Maid, Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10. The reversed order of the numeral cards in these suits looks rather odd, but all Italian trick-taking games of that time used this convention.

The Game

Assume there are N players. At the start of a game the cards are shuffled and dealt until there are fewer than N cards left — these excess cards play no further part in the game. The dealer then leads a card (places it face up on the table), and each player in turn plays a card face up on the preceding one. These N cards constitute a trick. The winner of a trick leads to the next trick. When playing to a trick one must follow suit (play a card of the same suit as the card that was led) if possible. If not, then one may trump the trick (play a trump card) or discard (play any card from one's hand). Also, if one holds the Idiot, one may play it (even if one could follow suit). There may be an advantage in doing this — see later. At the end of a game the cards are shuffled and the person to the left of the previous dealer deals the next hand and leads to the first trick.

For deciding who won a trick, the Idiot does not count as a trump; IT never wins a trick. If a trick does not contain any trump (except perhaps the Idiot) the highest ranking card of the suit led wins the trick, otherwise the highest ranking trump wins it. The player who won a trick takes all the cards that were played (except that the player who played the Idiot may swap it for another card), and gains the sum of the values of the cards captured (the value of the Idiot is used, not that of the card swapped for it).

Write a program that will read in a listing of cards in the order they were played and determine the scores of each player.

Input

The input consists of a description of a game setup, followed by the description of several games. The first line of the input consists of 22 letters giving the order of the trumps from highest to lowest. The Idiot (I) will always be the last letter. This will be followed by 86 integers on 5 lines giving the values of all the cards, firstly 22 integers on one line giving the value of the trumps in alphabetic order followed by 4 lines of 16 integers giving the values of the other cards in the order Cups, Coins, Swords and Batons. Within a suit the order is from King down to Ace. The next line contains only the number of players (*N*) in the range 2 to 8.

This will be followed by at least one but no more than nine games. Each game will consist of a sequence of card designators representing the cards in the order in which they are played, starting on a new line and terminating with ##. This sequence will be broken into lines of convenient length and not necessarily at the end of each trick. There will be zero or more spaces between card designators but there will not be any spaces at the end of a line. If the Idiot is played to any trick then the card designator following the *N*'th card of that trick represents the card that is swapped for the Idiot. This may be the Idiot, in which case there has not been a swap. Each deal will be terminated by ## and the entire set of games will be terminated by a line consisting of ##.

Output

For each game write a line containing the string '`Game `', followed by the number of the game, followed by ':' and the scores for each person in order in fields of width 7, assuming that the dealer for the first game is player 1. When all games have been played, write a summary line giving the total scores. Follow the format shown below.

Sample Input

```
XWVUTSRPONMLJHGFEDCBAI
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116
201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216
301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316
401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416
7
UT XT GT FT PT RT VT TT WT CT QS ET MT JT OT ST TS 2S DT LT IT 9C
AT NT DS TC NS 7S BT QN HT 9S KC 8S 6S PS PN MS NN IT TN 4S 5S 5B
KN KS 4N 5C 5N MN DN 7N AS 3N PB NC 8N AN 6N 4C PC DB QB QC 9N
2N 3C DC IT NB AC 6C IT MC 2C 8C 4B 8B MB 9B 7C TB 6B 2B 7B AB 3B ##
FT VT UT TT WT PT XT ET ST MS MT RT OT HT NS LT 2S JT NT CT BT
9S TS PN DT IT PS AT IT3S 7S KN 8S GT DS KS 8N 5S 9N QC QS DN AS
4N TN 5N PC 6S 7N 4S TC AN NC AC 3N 6N QN KC 2N MC 2C 9C QB MN
7C 6C DC PB 8C KB TB AB 4C NB 5B 5C 7B DB 8B MB 9B 3B 3C 4B 6B##
##
```

Sample Output

```
Game 1:    3004    105    371    677   1040   1544   8839
Game 2:    1517   2270   5799      0   4825      0   1768
Final :    4521   2375   6170    677   5865   1544  10607
```

Problem SP99-NZ4 Derivative Arithmetic

In mathematical modelling, it is often convenient to have the derivative d*f*/d*x* of a function as well as the function *f* itself. *Numerical differentiation* evaluates *f* twice, to compute *(f(x+v)-f(x))/v* for some small value of *v*. *Program differentiation* computes the derivative symbolically. *Derivative arithmetic,* on the other hand, works simply and efficiently without the problems involved in symbolic differentiation. For this we augment an expression with its derivative, i.e. we represent it by the ordered pair (*e*, d*e*/d*x*). Thus we have:

- A constant c is represented by (c, 0)
- The variable *x* is represented by (*x*, 1)
- Any other variable *y* is represented by (*y*, 0)
- (a, b) + (c, d) = (a+c, b+d)
- (a, b) – (c, d) = (a–c, b–d)
- (a, b) × (c, d) = (a×c, b×c+a×d)
- (0, b) ÷ (0, d) = (b÷d, 0) (using L'Hôpital's rule. The unknown derivative is approximated by zero)
  (a, b) ÷ (0, d) is undefined.
  (a, b) ÷ (c, d) = (a÷c, (b×c–a×d)÷c^2)
- $(a, b)^n$ = (a^n, n×b×a^{n-1}) for a ≠ 0 and n ≥ 1; otherwise it is undefined.
- sqrt (0, 0) = (0, 0)
  sqrt (0, b) is undefined.
  sqrt (a, b) = (√a, b÷(2√a)) if a > 0
- exp (a, b) = (exp a, b×exp a)
- ln (a, b) = (ln a, b÷a) if a > 0 otherwise it is undefined.

Write a program to simulate a 10-register HP-style (Reverse Polish) pocket calculator using this arithmetic. If any operation is undefined, clear the stack and continue from that point.

Input

Input will be a series of characters representing instructions chosen from the following set, where *n* represents a digit in the range 0 to 9, and *a* and *b* are numbers that may contain a decimal point and may start with a sign. Blanks and line breaks may be inserted arbitrarily to enhance readability, except within *a* and *b.*

(*a, b*)	Push this number onto the stack.
G*n*	Push the value of register *n* onto the stack. The ten registers are initialised to (0,0).
+, -, *, /	The topmost element of the stack is the right operand for this operator, and the one immediately below it is the left operand. Remove them, perform the operation, and push the result.
^*n*	Replace the topmost element (a, b) of the stack with $(a, b)^n$.
S, E, L	Apply sqrt, exp, or ln respectively to the topmost element of the stack.
P*n*	Remove the top element of the stack and store it in register *n*.
W	Remove the top element of the stack and write it on a new line. See sample output below
D	Push a copy of the top element of the stack onto the stack.
Q	Stop. This will always be the last character.

Output consists of one line for each W command, as described above. Numbers in the output should have **at least** six significant digits, may have leading and/or trailing blanks, and may, but need not, have an exponent part using either 'E' or 'e'.

Sample Input

```
(2,1)(2,1)*(2,1)/W(3,2)P6G6
(1,0)+P6G6^3 W (2,1)SP7G7W
G7ELWG7LEW G7DLDWEWLLDWEEWQ
```

Sample Output

```
(-2,1)
(64.0,96.0)
(1.41421,0.353553)
(1.41421,0.353553)
(1.41421,0.353553)
(0.346574,0.25)
(1.41421,0.353553)
(-1.05966,0.721348)
(1.41421,0.353553)
```

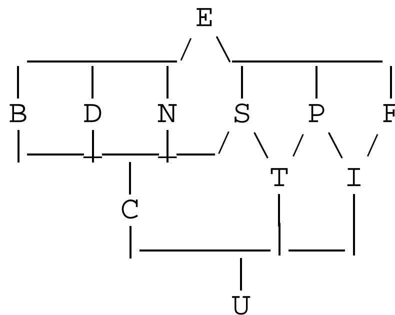
Problem SP99-NZ5 Type Inference

Compile time type-checking catches some errors, but the declarations can be a heavy notational burden. To obviate this, some programming languages, such as Lisp and Smalltalk, allow variables to take on values of any type. Others, such as SML and Haskell, enforce strict compile time types, but infer the type of a variable from the way it is used. This is done by initially assigning a general (unspecific) type to each variable and then constraining it to increasingly specific types as information about its usage is gathered.

Consider a simple programming language in which the types (from least to most specific) are:

U(nknown),
C(omparable),
T(extual),
I(ncomparable),
B(oolean),
D(ate),
N(umber),
S(tring),
P(attern),
F(ile), and
E(rroneous).

They can be arranged as follows, where there is a line running up from x to y if x is less specific than y.



As the program is parsed, the compiler accumulates constraints on the types of variables. For example, after seeing `x < y` the compiler concludes that:

tx >= ty (x's type is at least as specific as y's)
ty >= tx (y's type is as least as specific as x's)
tx >= C (x is comparable)
ty >= C (y is comparable)

For this programming language, the type inference problem is solved by finding the most general solution to a set of constraints such as this. There is always at least one solution (set every variable to E), the task is to find the most general one.

Input

Input will consist of a series of lines representing constraints, all applying to the same program, and terminating with a line containing the letter Q. The possible formats are:

`Vi>=t` where t is a type letter and i is a non-negative integer less than 1000,
`Vi>=Vj` where i and j are not necessarily distinct non-negative integers.

There will be no spaces in these lines. Each line except the Q line represents a single constraint. A type variable Vi will not appear until all the variables Vj ($0 \le j < i$) have appeared, so there will be no "gaps" in the variable numbering.

Output

Output will be one line for each distinct variable in the input, in the form shown below. The lines are to appear in ascending order of variable number *i*. The output represents the most general solution of the input constraints.

Sample Input

```
V0>=U
V1>=U
V0>=V1
V1>=V0
V0>=C
V1>=C
V1>=T
Q
```

Sample Output

```
V0=S
V1=S
```

Problem SP99-OZ1 Logical Line Counting

A software development company develops programs using a little known programming language with the following features:

- There are two styles of string. A string may be enclosed in 'single quotes' or in "double quotes". If a string is enclosed in double quotes, it can contain single quote characters, and if a string is enclosed in single quotes it can contain double quote characters. However, strings are not broken over a line, and cannot contain the same quote character used to enclose it.
- There are two styles of comment. All text from an @ character to the end of line is a line comment, and all text within ((double parentheses)) is a block comment.
- Block comments are not nested. All open parentheses inside a block comment are ignored. A block comment can extend over several lines.
- Comments cannot occur inside a string.
- Line comment characters or quotes inside a block comment have no significance.
- Double parentheses or quotes inside a line comment have no significance.
- The # character is guaranteed not to appear anywhere in a program; not even in strings or comments.
- A semi-colon is used to terminate statements and declarations. Every semi-colon not in a comment or a string is thus considered to terminate a logical line of the program.

A rough estimate for the size of a program can be made by counting logical lines, which means counting the semi-colons that do not appear inside a comment or string.

Write a program which reads text, counts the number of logical lines and the number of physical lines and issues warning messages for unterminated strings and unterminated block comments.

For each line containing an unterminated string, output a line of the form:

```
Unterminated string in line #.
```

where # is the physical line number for that program. Subsequent text must be handled as though the string had been terminated at the end of the line.

If the program contains an unterminated block comment, output a line of the form:

```
Unterminated block comment at end of file.
```

After error messages, if any, output a line of the form:

```
Program # contains # logical lines and # physical lines.
```

The input will be a series of programs each terminated by a # at the start of a line. The entire input will be terminated by a second # i.e. a line starting ##. There will be no other # characters in the input.

Output will be as specified above. It must have exactly the form shown, with no extra spaces around numeric output, and with a full stop at the end of each line and with no blank lines.

Sample Input

```
(( Block comment; ))
"string"; ('another string;')  @ line comment;
#
(((;(( A program typed by many monkeys; ));)))
yo mama! ;;@ ; '
' @ ;" ; ;'; "
)) ; (( ; )) '@"'; (";");
##
```

Sample Output

```
Program 1 has 1 logical lines and 2 physical lines.
Unterminated string on line 3.
Program 2 has 7 logical lines and 4 physical lines.
```

Problem OZ2 Juggling Trams

After many years of faithful service, the good city of Melbourne has decided to upgrade all of its aged green and gold trams to newer, more fashionable models. But alas, the pursuit of fashion has not been without its costs. The newer trams, whilst looking stunning, can only travel in straight lines. However this has not deterred the Melbourne City Council. In order to deal with this problem, they have laid more tracks and built more trams, so that there is now a vast grid of tram tracks criss-crossing throughout the city.

For simplification, consider the city to be a large grid of roads running north-south and east-west. Each road has a tram line running along it. You, the humble traveller, wish to travel south and west through this grid using the new tram system.

Trams run along each road at t-minute intervals, where t is a fixed time that has been determined by the Melbourne City Council. Thus, if you stand at any street intersection, you will see a tram travelling south every t minutes, and a tram travelling west every t minutes (although the trams will not necessarily pass by at the same times). We will assume that trams run at a constant speed, and take no time to pick up or drop off passengers.

Based on your training in city traversal optimisation theory, you wish to write a computer program that will find the fastest route from your starting point to your finishing point through the grid. You can hop on or off a tram at any intersection, and you may need to wait at some intersections for a tram to come along. You may assume that hopping on or off a tram takes no time at all, so, for instance, if a southward tram and a westward tram pass through an intersection at the same time, you may hop off one and immediately hop onto the other.

Input will consist of a number of data sets. The first line of each data set will contain two integers t and m, where t represents the number of minutes between each tram travelling down a street, as described earlier, with $1 \le t \le 60$ inclusive and m represents the number of minutes it takes a tram to move from one intersection to the next, with $m > 0$. Input will finish with $t = 0$ and $m = 0$.

The second line of the data set will contain two integers n and e, where n is the number of north-south streets and e is the number of east-west streets. Both n and e will be between 1 and 100 inclusive. North-south streets will be numbered from 1 to n, where 1 is the eastmost street and n is the westmost street. East-west streets will be numbered from 1 to e, where 1 is the northmost street and e is the southmost street.

The third line will contain four integers *sx sy fx fy*. Your journey must begin at the intersection of north-south street *sx* and east-west street *sy*, and must end at the intersection of north-south street *fx* and east-west street *fy*. You can assume that both intersections exist and that $sx \le fx$ and $sy \le fy$.

The fourth line will contain a single integer representing the number of minutes after midnight when your journey begins.

Following this will be n lines each containing two integers, *first* and k, where *first* is the number of minutes after midnight when the first tram leaves the northmost intersection along that street, and k is the total number of southward trams that will travel down that street throughout the day ($k > 0$). Note that these k trams will be spaced by intervals of t minutes. These lines are then followed by e lines containing the same information for the east-west streets.

Output consists of one line for each input data set. If it is possible to travel from the start point to the finish point, you must output a line of the form:

```
You arrive at hh:mm.
```

where `hh:mm` is the earliest time of day (using a 24-hour clock) at which you can arrive at your destination. Both hours and minutes must be two digits; use a leading 0 if necessary. You may assume that if a solution exists, the time of arrival will be before the next midnight.

If it is impossible to reach the destination, output the line:

```
Impossible.
```

Sample Input

```
30 3
5 4
2 2 5 4
93
30 5
100 6
115 4
100 7
30 8
10 10
0 11
20 9
10 10
30 3
5 4
2 2 5 4
300
30 5
100 6
115 4
100 7
30 8
10 10
0 11
20 9
10 10
0 0
```

Sample Output

```
You arrive at 01:52.
Impossible.
```

Problem SP99 OZ4 Australian Voting

The Australian electoral system elects Members of Parliament by holding local elections in a number of electoral divisions, with only one representative elected per division. Under this system each voter not only indicates his or her candidate of first preference by placing a number 1 on a box besides the name, but goes on to indicate who is their second choice, third choice and so on until all their preferences have been indicated. In order to be elected, a candidate must receive an absolute majority of the first preference votes cast in the electoral division, that is, more than 50% of the votes.

Counting consists of one or more rounds. If at any round any candidate receives strictly more than 50% of the first preference votes, then that candidate is duly elected; otherwise the candidate with the fewest first preference votes is excluded, and each vote currently assigned to this candidate is re-assigned as a first preference to the next most preferred candidate still remaining. This process continues until one candidate has more than half of the first preference votes cast and is duly elected.

If, at any round except the last, two or more candidates have the same number of first preferences, they are eliminated simultaneously and their votes transferred to the remaining candidates as before. If all the remaining candidates have the same number of first preference votes, the election is tied and a new ballot is called.

Consider 4 candidates labelled A, B, C and D and 11 voters (labelled for convenience) who vote as follows:

	A	B	C	D
v1:	1	2	3	4
v2:	1	4	2	3
v3:	1	3	4	2
v4:	1	2	4	3
v5:	4	1	2	3
v6:	3	1	4	2
v7:	2	1	3	4
v8:	3	1	4	2
v9:	4	2	1	3
vA:	4	3	1	2
vB:	3	4	2	1

First round: 4 3 3 1 -- no majority, D is eliminated. There is only one voting paper with D ranked 1 (vB) and that has C as a second candidate so C gets one more vote

Second round: 4 4 3 X - still no majority -- eliminate C. Take votes for C and reallocate v9 and vA to B, and vB to A.

Third round: 5 6 X X. Declare B the winner.

Write a program to determine the outcome of elections under this system.

Input will consist of a series of elections. The first round of line of each election will contain two integers C and V, where C is the number of candidates ($2 \le C \le 26$) and V is the number of voters ($1 \le V \le 1000$). Candidates are deemed to be labelled with an uppercase letter , starting with ‘A’, then ‘B’ and so on. (This is to preserve anonymity — this is a secret election after all). This will be followed by V lines with each line containing a voting preference in alphabetical order. These will be permutations of the numbers 1 to C. The file will be terminated with an election for which both C and V are zero.

Output will consist of one line for each election. If there is a clear winner output a line of the form:

```
Candidate X wins with n1 votes out of V.
```

otherwise a line of the form:

```
Election is a tie.
```

Sample Input

```
4 11
1 2 3 4
1 4 2 3
1 3 4 2
1 2 4 3
4 1 2 3
3 1 4 2
2 1 3 4
3 4 1 2
4 2 1 3
4 3 1 2
4 2 3 1
4 6
1 4 3 2
1 2 3 4
3 1 2 4
4 1 2 3
4 2 1 3
2 3 4 1
0 0
```

Sample Output

```
Candidate B wins with 6 votes out of 11.
Election is a tie.
```