2010/2011 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 1
Atlas Pagination

An atlas divides a map into separate pages in a process called *pagination.* Pagination starts with laying a rectangular grid over the area to be mapped. Some of the grid divisions will not contain an area to be mapped—these are excluded. The remaining grid divisions are assigned page numbers, proceeding from left to right and top to bottom.

An atlas is usable due to the references that direct the reader to the pages that hold adjacent portions of the map. Your team is to write a program that will produce these references.

The following steps describe how the references are assigned to the atlas pages:

- Overlay a rectangular grid on the area the be mapped.
- Exclude grid elements that do not contain any area to be mapped (see the shaded regions of Figure 1).
- Moving left-to-right, top-to-bottom, assign a sequential page number to each unshaded grid element, starting with page 1. For any grid page that contains an adjacent page, the corresponding margin must contain a reference to that adjacent page. In Figure 1, the reference in the left margin of page 13 is to refer the reader to page 12, and the bottom margin is to refer the reader to page 16. The top and right margins do not contain a reference.

Input to your program represents a series of rectangular grids used to overlay a map. The first line of each grid contains two integers, R and C, separated by exactly one space. R is the number of rows in the grid, and C is the number of columns in the grid. The following R lines describe the shading of the grid elements. The first character of the second line corresponds to the upper left element of the grid. A capital 'X' describes a shaded element, whereas a dot '.' indicates an unshaded element. R and C will be in the range 1–30 inclusive. There will always be at least one unshaded grid element. The last grid will be followed by the end-of-file.

Output for each map is a series of lines, one line per unshaded grid element (printed page). Each line must contain exactly five integers separated by single spaces, of the form

$p\ t\ r\ b\ l$

where p is the page number, and t, r, b, and l are the top, right, bottom, and left margin references of page p. If a particular margin is to remain empty (no adjacent grid page), print a page number of 0. Print page definitions in ascending page number order. No leading or trailing whitespace is to appear on an output line. Print an empty line after the information for the last page of each grid is printed.

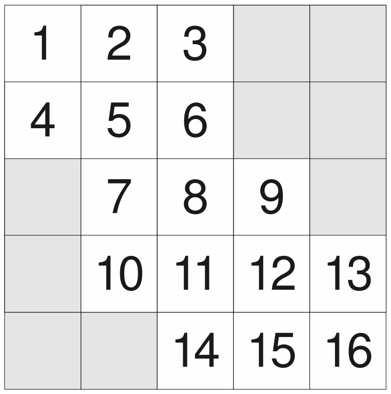

Figure 1. Diagram of the first sample input case.

Problem 1
Atlas Pagination (continued)

Sample Input

```
5 5
...XX
...XX
X...X
X....
XX...
2 4
X...
..XX
```

Output for the Sample Input

```
1 0 2 4 0
2 0 3 5 1
3 0 0 6 2
4 1 5 0 0
5 2 6 7 4
6 3 0 8 5
7 5 8 10 0
8 6 9 11 7
9 0 0 12 8
10 7 11 0 0
11 8 12 14 10
12 9 13 15 11
13 0 0 16 12
14 11 15 0 0
15 12 16 0 14
16 13 0 0 15

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

2010/2011 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 2
T-shirt Size Distribution

The staff of Swamp County College were well into their eleventh year of hosting a regional computer programming contest when they realized they were running out of time to prepare t-shirts for the contest day. For the previous ten years, the staff had always purchased shirts after the registration period closed, buying just enough shirts and assigning each school a shirt color from its school colors. Alas, this year the t-shirts had to be purchased prior to the closing of registration.

Although the staff could not predict which schools would enter, they could make a good estimate of projected attendance by the day they had to commit to a t-shirt contract. The staff decided that this year a single shirt color would be used, with the shirt order from the year with highest attendance as a good representation of the distribution of sizes. On the day the t-shirt order is due, the staff will project the expected attendance and scale the order according to the size distribution.

Your team is to write a program that takes representative distributions of shirt sizes, and given the projected attendance, produces t-shirt counts for each size.

Input to the program is a series of lines of text, each containing eight integers separated from each other by single spaces. The first seven values represent the number of t-shirts previously purchased, n_i, corresponding to sizes XS, S, M, L, XL, XXL, and XXXL, respectively. Summing the first seven values, $N = \sum n_i$, yields the total number of shirts previously ordered. The eighth value is P, the projected attendance. Input is terminated by the end-of-file.

Scale the t-shirt size counts,

$$p_i = \left\lfloor \frac{P n_i}{N} \right\rfloor$$

where the floor function $\lfloor x \rfloor$ gives the largest integer less than or equal to x. It is likely that the above computation of p_i will produce a shortage of t-shirts, $S = P - \sum p_i$, that is greater than zero. In this case, from smallest (XS) to largest (XXXL), start filling empty size counts ($p_i = 0$) with one shirt until $S = 0$ or there are no empty size counts. If there is still a shortage, purchase additional shirts at size XL to force $S = 0$.

For each line of input, print a line containing the integer values p_i, for sizes XS, S, M, L, XL, XXL, and XXXL respectively. Separate the values from each other by single spaces. No leading or trailing whitespace is to appear on an output line.

Sample Input

```
0 7 14 42 18 0 3 41
0 1 9 31 50 16 5 123
```

Output for the Sample Input

```
1 3 6 20 9 1 1
1 1 9 34 56 17 5
```

2010/2011 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 3
Manual Cipher

You are employed by a secret government agency that communicates with its field agents via encrypted messages only. The Agency is sending you to a location that does not have electricity, so you must send reports by scratching encrypted short messages onto tree bark and floating the bark down river for recovery by fellow agents. Five hours before departure, you consider it wise to provide a decryption program so that the Agency can decipher your important communications.

You will use a *numerically keyed aperiodic polyalphabetic cipher*. Such a scheme changes the encipherment alphabet after a specified number of letters, given in a numerical key. The numerical key describes how many characters to use in an encipherment alphabet before changing to a new alphabet. You will actually use a single alphabet, but rotate the substitution based upon the keyword. The decrypted (plain) alphabet consists of lowercase ASCII letters a–z and the space character. The encrypted (cipher) alphabet consists of uppercase ASCII letters A–Z and the space character.

Table 1 shows an encryption/decryption table based upon the codeword *BLUE* and numerical key *2314*. The plain-text alphabet is displayed across the top as column headers. The first column reading down spells out the codeword, BLUE. For each row, continue the cipher alphabet after the codeword character, cycling back to 'A' after the space character. Use the numerical key to construct the last column of the table. The numerical key does not become part of the encrypted message; it merely dictates how many characters to use from an encipherment alphabet before changing to the next alphabet.

Your program is to decrypt messages that were encrypted using this process: Given the plain-text message "watson are you there", use the first row to substitute two characters 'w' and 'a' with 'X' and 'B'. Next, use the second row to substitute the three characters 't' 's' 'o' with 'D' 'C' 'Z'. Substitute one character, 'n' with 'G'. Substitute the next four characters ' ' (space) 'a' 'r' 'e' with 'D' 'E' 'V' 'I'. Having exhausted the codeword and numerical keys ($2 + 3 + 1 + 4 = 10$ characters, change back to the first row, and substitute the two characters ' ' (space) 'y' with 'A' and 'Z'. Continue using the character counts 2–3–1–4–2–3–1–4–2–. . . until the entire message is enciphered.

Decryption is the reverse of encryption. Given the cipher text "XBDCZGDEVIAZZEKMLIVI", start at the first row and substitute two characters 'X' and 'B' with 'w' and 'a'. Change to the next alphabet on the second row and substitute three characters 'D' 'C' 'Z' with 't' 's' 'o'. Continue as with encryption, using the character counts 2–3–1–4–2–3–1–4–2–. . . until the entire message is deciphered.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	sp	
B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z		A	*2*
L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z		A	B	C	D	E	F	G	H	I	J	K	*3*
U	V	W	X	Y	Z		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	*1*
E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z		A	B	C	D	*4*

Table 1. Code word *BLUE* with a numerical key of *2314*. (sp indicates space)

Problem 3
Manual Cipher (continued)

Input consists of two or more lines. The first line contains the code word, a colon, and the numerical key. The codeword is built from the characters A–Z only (no spaces). A codeword may be 2–10 characters long. For each letter in the code word, there will be a corresponding decimal digit in the numerical key. '0' (zero) will never occur in the numerical key. The remaining lines contain enciphered text. Lines of enciphered text range between 1 and 79 characters. Input is terminated by end-of-file.

The plain-text alphabet is always a–z and space, and the cipher text alphabet is always A–Z and space. For each line of enciphered text, your program is to produce a line of deciphered plain-text. Each plain-text output line is to be the same length as the corresponding enciphered input line.

Sample Input

```
BLUE:2314
XBDCZGDEVIAZZEKMLIVI
NPCAEBXSIWAVYMPUVEFPFACPYXDRIXUJYR
CBBVKKYRRMOHKWZPDWIREAXZBYDXVIFT
```

Output for the Sample Input

```
watson are you there
mosquitoes unbearable send netting
bark running low send more trees
```

2010/2011 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 4
Musical Road

There is an unusual road in Lancaster, California—it acts as a musical instrument! The road has grooves cut across it. As a car drives over the grooves the resulting vibrations are transmitted to the car. If the vibrations are at appropriate frequencies, musical notes at standard frequencies are formed. By cutting the proper number of grooves in a road with the proper distance between them, a car driven over the road at the proper speed will literally play a tune. The road in Lancaster is supposed to play the beginning of the "William Tell Overture."

Each groove cut into the road causes one vibration as the tires cross over it. By crossing the proper number of grooves per second, a note is played. For example, A below middle C has a frequency of 220 Hz. In order to play this note for one second, the car must cross 220 grooves in that second.

There are twelve notes on the musical scale, listed below in order of increasing frequency:
C C♯/D♭ D D♯/E♭ E F F♯/G♭ G G♯/A♭ A A♯/B♭ B

The seven natural notes are represented by the upper-case letters A through G. The other five notes are referred to by using either a note letter followed by a sharp symbol (♯—meaning one note higher), or a note letter followed by a flat symbol (♭—meaning one note lower). We will use '#' to represent the sharp symbol and lower case 'b' to represent the flat symbol.

An *octave* is made up of the twelve notes listed above, from C to B. The note above B is C in the next octave. A note is represented by its letter followed by an optional sharp or flat symbol and its octave number. Octaves are represented by single digits.

The frequency of a given note is a factor of $\sqrt[12]{2}$ higher than the previous note. Frequencies are determined by using the A note in octave 4 as a base. A4 has a frequency of 440 Hz. Therefore, A♯4 has a frequency of $(440 \times \sqrt[12]{2})$ Hz, and A3 has a frequency that is half of A4 (220 Hz). The musical road uses octaves 2 through 5.

Each note has a duration associated with it. These durations are referred to as full, half, quarter, or eighth notes. (A half note is played for half the duration of a full note, a quarter note for a quarter of the duration of a full note, and so on.) The specifications for the musical road will give the number of quarter notes to be played per minute. The duration of a note can be increased by 50% by placing a dot ('.') after the note—so a dotted half-note would be played for 75% as long as a full note.

Rests occur in a song when no note is playing. No grooves would be cut in the road for the duration of a rest. Rest durations are also measured in terms of whole, half, quarter, and eighth notes, and can also be dotted to increase their duration by 50%.

Your team is to write a program that will, given a target road speed along with the notes and tempo for a song, will produce the required groove counts and distance between grooves needed for each note or rest in the song.

Input to your program will be a series of songs ending with the end-of-file. The input for each song will be as follows: The first line will contain the targeted road speed in miles per hour as an integer, followed by a single space and the integer tempo in quarter notes per minute. The remaining lines will contain notes to be played on the road. Each note will be given as a capital letter, an optional sharp ('#') or flat ('b') sign, a single digit representing the octave, a single digit representing the note duration (1, 2, 4, or 8 for full, half, quarter, or eighth note respectively), and an optional dot. Rests will be represented by the letter 'R', a note duration digit, and an optional dot. Notes will be separated from each other by one or more spaces. Each song will end with an empty line.

Problem 4
Musical Road (continued)

For each note, your program is to print a line containing the number of grooves rounded to the nearest integer and the distance between the grooves in feet and inches to the nearest 1/8 inch. For a rest, print a groove count of zero followed by the proper distance. The rounding of the groove count and distance are to occur after both values are computed.

Each output line is to be printed in the following format:

- The output line is to begin with the groove count followed by a colon.
- The distance is to be printed after the colon using the following format, where the components in square brackets are only to be printed if needed:
  [(space)(whole feet)'][(space)(whole inches)][(space)(fractional inches)]["]
- Omit any elements that have zero values.
- Fractions are to be printed in reduced form.
- No trailing space is to appear on an output line.
- Print an empty line at the end of each song (including the last).

If two identical notes appear in the input, print the groove count and spacing for each note—your program is not to consider the spacing between notes.

Hint: There are 5280 feet in a mile.

Sample Input

```
40 120
F44 G48 E48 F44 G44  A44 Bb48 A48

60 120
E42. D42. C42. C42 R4
E42. D42.
```

Output for the Sample Input

```
175: 2"
98: 1 3/4"
82: 2 1/8"
175: 2"
196: 1 3/4"
220: 1 5/8"
117: 1 1/2"
110: 1 5/8"

494: 3 1/4"
440: 3 5/8"
392: 4"
262: 4"
0: 44'
494: 3 1/4"
440: 3 5/8"
```

2010/2011 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 5
Triangular Redistricting

The time has come for Triangle City to re-draw the city council district boundaries. Triangle City is so named because the city limits form a triangle. The city is not large (its population varies, but has never exceeded 30,000), so the city council has only three members. Fortunately, the population of the city is a multiple of three.

The last redistricting was very political and resulted in gerrymandered districts. In order to prevent that from happening again, the city charter now requires that the three districts not only contain the same number of people, but that they be split on a single point. The district boundaries will be formed by connecting the splitting point to each vertex of the boundary triangle, resulting in three triangular council districts.

Your team is to write a program that, given the boundaries of the city and the locations of the residents on a Cartesian grid, will find a suitable single splitting point.

Input data for your program will be in the following format:

- Line 1: An integer n, $3 \leq n \leq 30000$, denoting the number of people. n is always a multiple of 3.
- Lines 2 through 4: Each line will have two floating-point numbers x_i and y_i denoting the coordinates of the ith corner of the city, separated by a single space. $-10 \leq x_i, y_i \leq 10$. The three corners are specified in counterclockwise order.
- Lines 5 through $(n+4)$: Each line will have two floating-point numbers x_j and y_j denoting the coordinates of the jth person, separated by a single space. $-10 \leq x_j, y_j \leq 10$. All people lie in the interior of the city triangle.

The input stream ends with the end-of-file.

Your program is to print a single line with two floating-point numbers x and y denoting the coordinates of the splitting point, separated by a single space and with no leading or trailing whitespace.

Each person must lie in the interior of one of the three generated triangles. Assume a person is a single point with zero radius, and that a wall is a line with zero thickness. It is guaranteed that there exists a splitting point. It is not guaranteed that such a splitting point is unique; any splitting point that partitions the people into three sets of equal size will be judged correct. You must print the answer with sufficient precision that the judges' validation program will be able to determine that the point is correct.

Sample Input

```
3
0.0 0.0
10.0 0.0
0.0 10.0
4.0 4.0
1.0 4.0
4.0 1.0
```

Output for the Sample Input

```
2.7142857137648977 2.714285714850612
```

2010/2011 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 6
Module Madness

While going through the Swamp County College archives some strange drawings were discovered. They appear to specify some sort of computing device and seem to be written in a predecessor of today's relay logic ladder diagrams which are used for Programmable Logic Controllers. Luckily it isn't necessary to understand relay logic to understand them since they can easily be translated to boolean equations. Your job is to read in the diagram of a module and a list of input states, simulate the module, and produce the output each time the input changes.

See Figure 1 for a simple diagram and the equivalent boolean equation.

The components of a diagram are gates (relay contacts) and gate controllers (relay coils). The diagrams are called ladder logic because you can picture the ladder rails as the power buses at each side of the diagram and the rungs as connected lines with one and only one connection to each bus on the left and right. The controllers can have no other components (gates or controllers) between them and the right hand bus, only wires. All gates are to the left of the controllers. Gates in series are an AND circuit and gates in parallel are an OR circuit. A controller is set to the results of the equations determined by the gates to its left. In turn, its associated gates are set by the state of the controller.

Input gates are set either by the outputs of another module or some input device. Output controllers can set the inputs of another module, control an output device, or control their associated gates in the diagram. Relay controllers may only control their associated gates in the diagram.

The rules for the character representation of the relay ladder logic diagrams follow:

- The left and right rails are not drawn. Wires are drawn with '-' characters. There will always be at least one '-' between components.
- Gates (which include input, relay, and output gates) are drawn enclosed in square brackets. There are up to ten possible input gates X0–X9, up to ten possible output gates Y0–Y9, and up to 100 possible relay gates R00–R99. Each of these may appear multiple times in the diagram and in any rung. They take either the boolean value of the associated input or controller, for example, [R18] or its logical negation [!R18]. The numbering is arbitrary—for example, a given diagram may only use X2, X5, and X8 as inputs.
- Gates can be connected in series, parallel, or a combination.
- Controllers appear in the diagram enclosed in parentheses. There are up to ten possible output controllers Y0–Y9 and up to 100 possible relay controllers R00–R99. A given controller may appear only once in a diagram. A controller will appear if its corresponding gate appears. However, a controller may appear without any corresponding gate. Controller numbering is arbitrary.
- Only controllers can connect to the right rail. Every rung has at least one controller connecting to the right rail. Controllers may not appear in series, only in parallel on a rung. If controllers appear in parallel they each take the value determined by the boolean equation of all the gates to the left of their common junction on the rung.
- Connections can only be made to adjacent lines within a rung. A connection to a lower line is indicated by 'U' and to an upper line by 'L'. These indicators must line up properly on the two lines. There will be at least one '-' between a connector and a component.

Problem 6
Module Madness (continued)

The diagram is drawn so logic always flows to the right, assuming the left rail is the source, or up or down one line at one of the connector pairs.

These diagrams do not have any race or other anomalous conditions, such as "`---[!R0]---(R0)---`", so the controllers are guaranteed to settle into a stable state in a short time but it is very likely several iterations will be needed before they do so.

On initial power up, all inputs and controllers are off, which is logical 0.

The input diagram is a series of lines of at most 72 characters, terminated by an empty line. To save typing, a line ending in "`->`" is considered to extend to the right rail. There are at most 100 lines in the diagram.

The test inputs follow, one case per line, terminated by end of file. Each test case consists of exactly ten binary digits ('0' for off and '1' for on) with the left digit being X0 and the rightmost digit being X9. Any input not appearing in the diagram is ignored. There are at most 100 test inputs.

Your program is to first print the outputs after the power on transients have settled, then apply each input in turn and print the outputs when they have settled. Note that this means applying each input to the state resulting from the previous test case, not power cycling the module between cases. Also note that the execution order is not specified, since relays run on their own time.

Print the outputs for each test case as exactly ten characters on one line with the left character being Y0 and the rightmost being Y9. Print '0' for off, '1' for on, and 'X' if the output does not appear in the diagram.

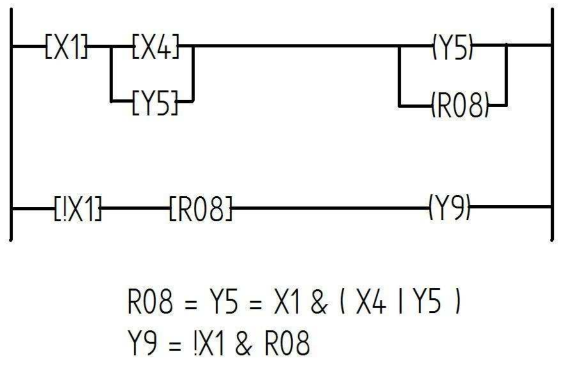

Figure 1. Sample ladder rungs and corresponding boolean equations.

```
----[X1]---U----[X4]----U-------U----(Y5)----U------>
           L----[Y5]----L       L---(R08)----L
----[!X1]-----------------[R08]------(Y9)----------->
```

Figure 2. Text representation of the diagram in Figure 1.

Problem 6
Module Madness (still continued)

Sample Input

```
---U---[!X3]---[X1]--U-----(Y2)-->
   L---[X3]----[!X1]-L
-----[X3]---[X1]-------U--(R00)-U-->
                       L--(Y9)--L
          U---[!X0]--[!X2]---[R00]--U
----U-----L----[X0]---[X2]---[R00]--L------U----------(Y1)-->
    L-U--------[X0]--[!X2]--[!R00]-----U---L
      L-------[!X0]---[X2]--[!R00]-----L
--U----[X0]----[X2]------------------U-------------(Y0)-->
  L-----[R00]-------U-----[X0]----U--L
                    L-----[X2]----L
---[X9]------U----[Y0]----U---(Y8)--->
             L----[Y8]----L

0000000000
0001000000
0010000000
0011000000
0100000000
0101000000
0110000000
0111000000
1000000000
1001000001
1010000001
1011000001
1100000001
1101000001
1110000001
1111000001
0000000001
0000000000
```

Output for the Sample Input

```
000XXXXX00
000XXXXX00
001XXXXX00
010XXXXX00
011XXXXX00
001XXXXX00
010XXXXX01
011XXXXX00
100XXXXX01
010XXXXX00
011XXXXX00
100XXXXX10
101XXXXX10
011XXXXX10
100XXXXX11
101XXXXX10
110XXXXX11
000XXXXX10
000XXXXX00
```

2010/2011 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 7
Zombie Blast!

Help!! The zombies are marching! The zombie invasion has begun, and their legion is on the battlefield, coming toward our last line of defense.

All hope is not lost, though. In anticipation of the forthcoming doom, you have deployed a host of Adjustable Conflagration Mines (ACMs) on the battlefield. You must act quickly and have time only to configure a single blast radius that will be set for all the mines. You then must detonate all mines simultaneously. Each mine will instantly incinerate any zombie within its blast radius.

You take a satellite image to give you a map of the situation. The map is a rectangular region divided into square cells with a side length of one unit. Each cell is either empty, occupied by a zombie, occupied by a mine, or occupied by both a zombie and a mine.

A zombie will be incinerated by a mine if the distance between the center of the cell it occupies and the center of the mine's cell is less than or equal to the blast radius. A blast radius of zero will incinerate a zombie if it is in the same cell as the mine.

In order to minimize collateral damage, you have to set off the mines with the smallest blast radius that will still incinerate all the zombies. Your team is to write a program that will read satellite maps and find the blast radius for each map.

Each satellite map will be represented by a series of lines, with characters representing the occupancy of each cell. Four characters will be used to represent the occupancy of each cell:

- 'Z' denotes a cell with a zombie,
- 'M' denotes a a cell with a mine,
- 'B' denotes a cell with both a zombie and a mine, and
- '.' denotes an empty cell.

Input to your program will be a series of satellite maps terminated by end-of-file. Each map except the last will end with an empty line. The last map will end with the end-of-file.

There will be at most 2,000 lines in a single map. All lines on a given map will contain the same number of cells. No map line will contain more than 2,000 cells. Every map will have at least one zombie and one mine on it. There will be at most 160,000 mines and 160,000 zombies in each map.

For each map, your program is to print the smallest blast radius on a single line, rounded to two decimal places, with no leading or trailing whitespace or unnecessary zeroes.

Sample Input

```
M..Z
..ZZ
B..Z

.ZZ.M
Z.Z..
.Z.ZZ
Z.Z.Z

...M.....
.........
.M.......
.........
.........
.........
.........
.........
...Z.....

B
```

Output for the Sample Input

```
3.16
5.00
6.32
0.00
```

2010/2011 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 8
Resistor Substitution

Swamp County College has joined the TinySat consortium which will design and fly a series of small satellite experiments. You are on the team that is designing the electronics package. Electronic parts that can withstand the stresses of space travel are known as *space qualified* parts. Space qualified parts are rugged and made to extremely tight tolerances. While extremely accurate, these parts are also very expensive so you have access to an inventory with a limited number of component values.

The circuits you are building include resistors with values that will not be known until after the actual circuit is built. The needed resistor values will be determined by calibration using variable resistors which are not space qualified. There will be space left on the circuit board for up to three space qualified fixed resistors in various configurations to substitute for each variable resistor. These configurations are shown in Figure 1.

The resistance values R for each configuration (in order from simplest to most complex) are:

$A : R = R1$	Single resistor
$B : R = R1 + R2$	Two resistors in series
$C : R = \frac{1}{1/R1+1/R2}$	Two resistors in parallel
$D : R = R1 + R2 + R3$	Three resistors in series
$E : R = \frac{1}{1/R1+1/R2+1/R3}$	Three resistors in parallel
$F : R = R1 + \frac{1}{1/R2+1/R3}$	Three resistors in series-parallel

Resistor values are real numbers possibly followed by 'k' or 'm' (upper or lower case). A 'k' means multiply the value by 1,000 and 'm' by 1,000,000.

Your team is to write a program that will find a close enough match to a specified variable resistor value using the space qualified resistor values in the inventory in one of the configurations listed above.

Input to your program will begin with a series of lines containing the space qualified resistor values in the inventory. Each line will contain one or more resistor values, separated from each other by whitespace. The values are in no particular order. Each value appears only once. You may use the same value multiple times in a solution. The inventory is terminated by an empty line.

Desired target resistor values and the tolerance of a match follow, one case per line, terminated by end of file. The resistor value is first and is separated by whitespace from the tolerance.

The tolerance is a real number expressed as a percentage—a value of 10 means 10%. If the target variable resistor value is 1000 with a tolerance of 10, then any value for R in the range 900–1100 would be considered a match.

Resistor values in the inventory as well as those to be matched will be in the range 1 to 5,000,000. The tolerance values will range from 0.0001% to 25%.

If there are matches in several configurations, choose the simplest configuration. Note that this means that if there is a good enough solution in configuration C, for example, choose that one even if there is a better solution in, say, E.

Within a configuration find the closest solution. If there are several equally good solutions within a configuration, choose the one with the smallest $R1$. If there are multiple equally good three-resistor solutions with the same $R1$, choose the one with the smallest $R2$.

Problem 8
Resistor Substitution (continued)

For each test case that has a close enough solution, print the solution on a line as follows: the configuration letter; a single space; and the selected inventory resistor values separated by single spaces, without leading or trailing spaces. Order the resistor values: $R1, R2, R3$. If there is no close enough solution, print the string "no result" on a line by itself. Print the resistor values with three digits and at least one significant digit before any decimal point, followed by 'K' or 'M' (upper case only) if necessary.

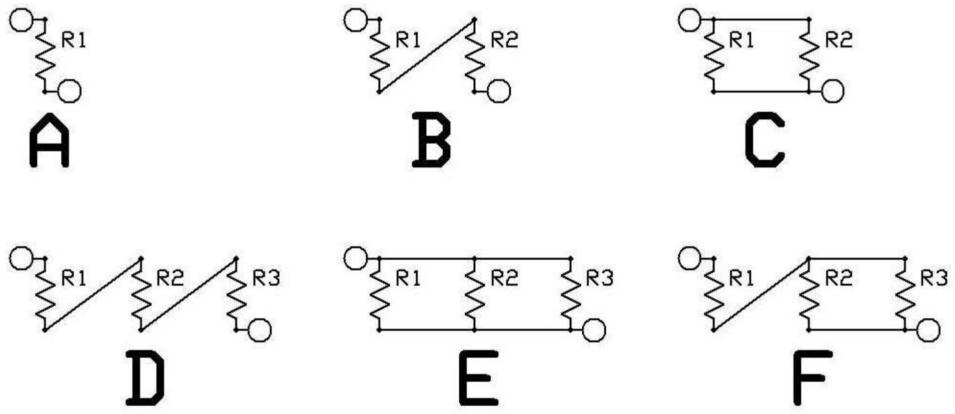

Figure 1. Schematic diagrams of the space qualified resistor combinations.

Sample Input

```
10   15   22   33   47   68   100  150  220  330  470  680
1.0k 1.5k 2.2k 3.3k 4.7k 6.8k 10k  15k  22k  33k  47k  68k
100k 150k 220k 330k 470k 680k 1.0m 2.0m

84.3k .1
84.3k 1
517 .1
2.2K .1
6.12k .1
6.12k .5
3.47M .1
```

Output for the Sample Input

```
no result
D 1.50K 15.0K 68.0K
B 47.0 470
A 2.20K
no result
E 6.80K 68.0K 680K
D 470K 1.00M 2.00M
```