Problem A: RGB (blue balloon)

A color reduction is a mapping from a set of discrete colors to a smaller one. The solution to this problem requires that you perform just such a mapping in a standard twenty-four bit RGB color space. The input consists of a target set of sixteen RGB color values, and a collection of arbitrary RGB colors to be mapped to their closest color in the target set. For our purposes, an RGB color is defined as an ordered triple (R,G,B) where each value of the triple is an integer from 0 to 255. The distance between two colors is defined as the Euclidean distance between two three-dimensional points. That is, given two colors (R_1,G_1,B_1) and (R_2,G_2,B_2), their distance D is given by the equation

$$D = \sqrt{(R_2 - R_1)^2 + (G_2 - G_1)^2 + (B_2 - B_1)^2}\ .$$

Standard Input

The input is a list of RGB colors, one color per line, specified as three integers from 0 to 255 delimited by a single space. The first sixteen colors form the target set of colors to which the remaining colors will be mapped. The input is terminated by a line containing three –1 values.

Standard Output

For each color to be mapped, output the color and its nearest color from the target set.

Sample Input	Sample Output

```
0 0 0
255 255 255
0 0 1
1 1 1
128 0 0
0 128 0
128 128 0
0 0 128
126 168 9
35 86 34
133 41 193
128 0 128
0 128 128
128 128 128
255 0 0
0 1 0
0 0 0
255 255 255
253 254 255
```
	```
	(0,0,0) maps to (0,0,0)
	(255,255,255) maps to (255,255,255)
	(253,254,255) maps to (255,255,255)
	(77,79,134) maps to (128,128,128)
	(81,218,0) maps to (126,168,9)
	```

```
77 79 134
81 218 0
-1 -1 -1
```

Problem B: Morse

(red balloon)

Morse code represents characters as variable length sequences of dots and dashes. In practice, characters in a message are delimited by short pauses. The following table shows the Morse code sequences:

A	.-	H	....	O	---	V	...-
B	-...	I	..	P	.--.	W	.--
C	-.-.	J	.---	Q	--.-	X	-..-
D	-..	K	-.-	R	.-.	Y	-.--
E	.	L	.-..	S	...	Z	--..
F	..-.	M	--	T	-		
G	--.	N	-.	U	..-		

Note that four dot-dash combinations are unassigned. For the purposes of this problem we will assign them as follows (these are not the assignments for actual Morse code):

underscore	..--	period	---.
comma	.-.-	question mark	----

Thus, the message "ACM_GREATER_NY_REGION" is encoded as:

```
.- -.-. -- ..-- --. .-. . .- - . .-. ..-- -. -.-- ..-- .-. . --. .. --- -.
```

M.E. Ohaver proposed an encryption scheme based on mutilating Morse code. Her scheme replaces the pauses between letters, necessary because Morse is a variable-length encoding that is not prefix-free, with a string that identifies the number of dots and dashes in each. For example, consider the message ".--. ". Without knowing where the pauses should be, this could be "ACM", "ANK", or several other possibilities. If we add length information, however, ".--. 242", then the code is unambiguous.

Ohaver's scheme has three steps, the same for encryption and decryption:

1. Convert the text to Morse code without pauses but with a string of numbers to indicate code lengths
2. Reverse the string of numbers
3. Convert the dots and dashes back into to text using the reversed string of numbers as code lengths

As an example, consider the encrypted message "AKADTOF_IBOETATUK_IJN". Converting to Morse code with a length string yields ".--.-.--..----..-...--..-...---.-.--..--.-..--...----.232313442431121334242". Reversing the numbers and decoding yields the original message "ACM_GREATER_NY_REGION".

This problem requires that you implement Ohaver's encoding algorithm.

Standard Input

The input will consist of several messages encoded with Ohaver's algorithm. The first line of the input is an integer *n* that specifies the number of test cases. The following *n* lines contain one message per line. Each message will use only the twenty-six capital letters, underscores, commas, periods, and question marks. Messages will not exceed 100 characters in length.

Standard Output

For each message in the input, output the line number starting in column one, a colon, a space, and then the decoded message. The output format must be adhered to precisely.

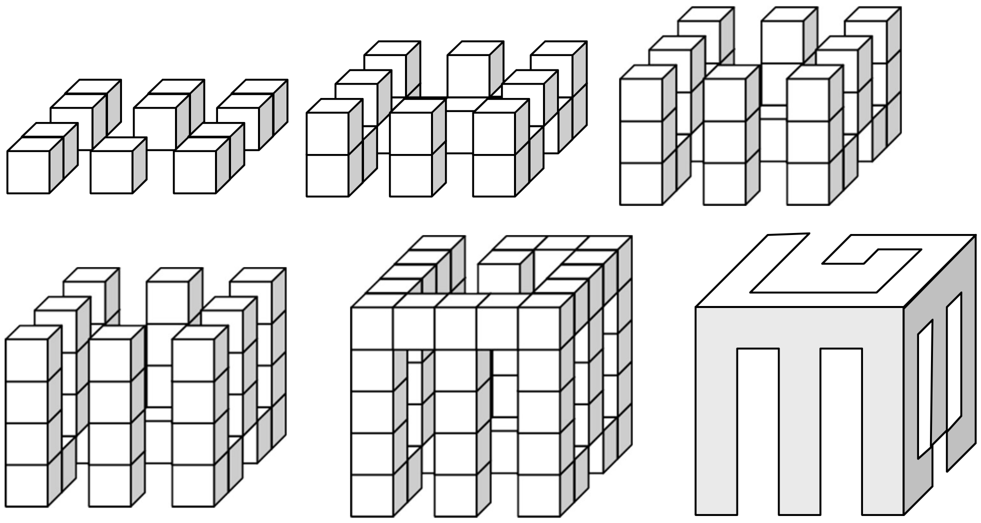
Sample Input	Sample Output
5 AKADTOF_IBOETATUK_IJN PUEL QEWOISE.EIVCAEFNRXTBELYTGD. ?EJHUT.TSMYGW?EJHOT DSU.XFNCJEVE.OE_UJDXNO_YHU?VIDWDHPDJIKXZT?E	1: ACM_GREATER_NY_REGION 2: PERL 3: QUOTH_THE_RAVEN,_NEVERMORE. 4: TO_BE_OR_NOT_TO_BE? 5: THE_QUICK_BROWN_FOX_JUMPS_OVER_THE_LAZY_DOG

Problem C: Blocks

(yellow balloon)

Plato believed what we perceive is but a shadow of reality. Recent archaeological excavations have uncovered evidence that this belief may have been encouraged by Plato's youthful amusement with cleverly-designed blocks. The blocks have the curious property that, when held with any face toward a light source, they cast shadows of various letters, numbers, shapes, and patterns. It is possible for three faces incident to a corner to correspond to three different shadow patterns. Opposite faces, of course, cast shadows which are mirror images of one another.

The blocks are formed by gluing together small cubes to form a single, connected object. As an example, the figures below show, layer by layer, the internal structure of a block which can cast shadows of the letters "E", "G", or "B".



Only a partial set of blocks was discovered, but the curious scientists would like to determine what combinations of shadows are possible. Your program, the solution to this problem, will help them! The program will input groups of shadow patterns, and for each group will report whether or not a solid can be constructed that will cast those three shadows.

Standard Input

The input contains a sequence of data sets, each specifying a dimension and three shadow patterns. The first line of a data set contains a positive integer $1 \leq n \leq 20$ that specifies the dimensions of the input patterns. The remainder of the data set consists of 3*n* lines, each containing a string of *n* "X" and "-" characters. Each group of *n* lines represents a pattern. Where an "X" appears a shadow should be cast by the final solid, and where a "-" appears, light should pass through. For this problem, the input patterns may be assumed to have at least one "X" along each edge of the pattern. The input is terminated by a line containing a single zero in place ofa valid dimension.

Standard Output

For each data set in the input, output the data set number and one of the following messages:

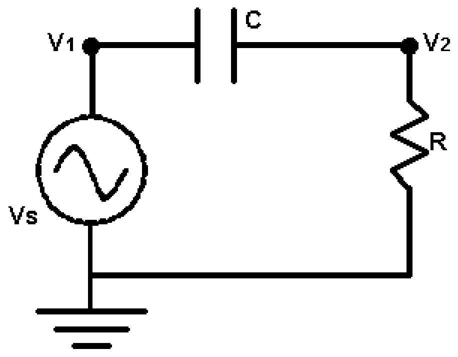
```
Valid set of patterns
Impossible combination
```

For a set of patterns to be considered valid, it must be possible to construct, by gluing unit cubes together along their faces, a one-piece solid capable of casting the shadow of each of the input patterns.

Sample Input	Sample Output
5 XXXXX X---- X--XX X---X XXXXX XXXXX X---- XXXXX X---- XXXXX XXXXX X---X XXXX- X---X XXXXX 3 X-- -X- --X XX- XXX -XX -XX XXX XX- 0	Data set 1: Valid set of patterns Data set 2: Impossible combination

Problem D: Plot (Orange balloon)

Consider the AC circuit below. We will assume that the circuit is in steady-state. Thus, the voltage at nodes 1 and 2 are given by $\mathbf{v_1 = V_S \cos \omega t}$ and $\mathbf{v_2 = V_R \cos (\omega t + \theta)}$ where $\mathbf{V_S}$ is the voltage of the source, ω is the frequency (in radians per second), and **t** is time. $\mathbf{V_R}$ is the magnitude of the voltage drop across the resistor, and θ is its phase.



You are to write a program to determine $\mathbf{V_R}$ for different values of ω. You will need two laws of electricity to solve this problem. The first is Ohm's Law, which states $\mathbf{v_2 = i R}$ where **i** is the current in the circuit, oriented clockwise. The second is $\mathbf{i = C \, d/dt \, (v_1 - v_2)}$ which relates the current to the voltage on either side of the capacitor. "**d/dt**" indicates the derivative with respect to **t**.

Standard Input

The input will consist of one or more lines. The first line contains three real numbers and a non-negative integer. The real numbers are $\mathbf{V_S}$, **R**, and **C**, in that order. The integer, ***n***, is the number of test cases. The following ***n*** lines of the input will have one real number per line. Each of these numbers is the angular frequency, ω.

Standard Output

For each angular frequency in the input you are to output its corresponding V_R on a single line. Each V_R value output should be rounded to three digits after the decimal point.

Sample Input	Sample Output
1.0 1.0 1.0 9	0.010
0.01	0.032
0.031623	0.100
0.1	0.302
0.31623	0.707
1.0	0.953
3.1623	0.995
10.0	1.000
31.623	1.000
100.0	

Problem E : Logic

(purple balloon)

For this problem you will determine the output of a logic circuit composed of one or more inputs, zero or more dual-input AND/OR gates, and one output. The input circuits are drawn with standard ASCII characters. Circuit paths are represented using horizontal and vertical lines, and junctions. Horizontal lines are represented with dash characters (ASCII code 45 decimal), vertical lines with vertical bar characters (ASCII code 124 decimal), and junctions with plus characters (ASCII code 43 decimal). Inputs are represented using the capital letters A through Z, and the output is represented by a question mark. AND and OR gates are represented as shown in the leftmost entries in the figure below, and their orientation will always be exactly as shown. The location of the gate inputs and output is shown by the middle figure below. Finally, gate inputs or its output can be inverted, represented by a lowercase "oh" character (ASCII code 111 decimal) at the input or output location. The figure on the right below shows a simple but complete logic circuit.

```
:\
: )
:/
```

AND gate

```
:\
: >
:/
```

OR gate

```
-:\
 : )-
-:/
```

Gate with inputs

```
-o:\
  : )o-
--:/
```

An inverted top input and an inverted output

```
A-o:\
   : )o-?
B--:/
```

Two logic inputs and the output

Circuits in the input will obey the following guidelines:

1. The maximum size of the circuit picture is 100 by 100 characters.
2. A path always travels in a straight line unless altered by a junction. At a junction, the path can and will make a ninety degree turn. Two junctions will not be horizontally or vertically adjacent.
3. No paths will be "broken". That is, every path character is guaranteed to be adjacent on both sides to either another path character of the same type, a junction, a gate input, a gate output, a logic input, or the logic output.
4. Circuit paths do not cross or intersect other paths.
5. Gate inputs always approach horizontally from the left as shown above. Gate outputs always leave horizontally to the right as shown above.
6. Inversions may only appear immediately adjacent to a gate input or output, and will always be preceded (in the case of an input) or followed (in the case of an output) by at least one dash as shown above.

Standard Input

The end of a logic diagram in the input is indicated by line containing only a single asterisk in the first column. Following this are several lines which indicate the state of the inputs in the logic diagram. Each of these lines is a string of twenty-six "0" (zero) or "1" characters, with the first position representing the state of input A, the second position representing the state of input B, etc. Note that input values which are not actually used in the circuit may simply be ignored. The list of input states is terminated by a line containing only a single asterisk character in the first column.

Following the asterisk which terminates the list of input states is another circuit diagram followed by a list of input states, which is then followed by another circuit diagram and list of input states, and so on. The end of the standard input is a single line with two consecutive asterisk characters. The standard input will always contain at least one circuit and one set of inputs for that circuit.

Standard Output

The program is to report the value of the output (0 or 1) of each logic circuit, one value per line, for each set of input values in the list which follows the circuit. The list of outputs for each circuit should be separated by a single blank line.

Sample Input

```
A---:\
    : )---?
B---:/
*
00000000000000000000000000
10000000000000000000000000
01000000000000000000000000
11000000000000000000000000
*
A---+
    |
    +---:\
        : >o---:\
    +---:/     : )---?
    |      C--o:/
B---+
*
00000000000000000000000000
11100000000000000000000000
*
**
```

Sample Output

```
0
0
0
1

1
0
```

Problem F : Microprocessor

(green balloon)

Consider a small microprocessor that has the following properties:

- Each word is four bits.
- Addresses are two words. The high word always comes first. That is, the high word of a two-word address will always occupy the lower word of memory.
- Memory is 256 words.
- There are two accumulators, A and B, each storing one word.
- There are nine instruction codes. Each instruction requires at least one word to store the code that specifies the instruction. Four instructions have arguments and require an additional two words.

Each 4 bit number can have the values from 0 to 15, inclusive, in base 10. We will write these using hexadecimal in the usual way, i.e. A means 10, B means 11, etc.

These are the nine instructions:

Code	**Words**	**Description**
0	3	LD: Load accumulator A with the contents of memory at the specified argument.
1	3	ST: Write the contents of accumulator A to the memory location specified by the argument.
2	1	SWP: Swap the contents of accumulators A and B.
3	1	ADD: Add the contents of accumulators A and B. The low word of the sum is stored in A, and the high word in B.
4	1	INC: Increment accumulator A. Overflow is allowed; that is, incrementing F yields 0.
5	1	DEC: Decrement accumulator A. Underflow is allowed; that is, decrementing 0 yields F.
6	3	BZ: If accumulator A is zero, the next command to be executed is at the location specified by the argument. If A is not zero, the argument is ignored and nothing happens.
7	3	BR: The next command to be executed is at the location specified by the argument.
8	1	STP: Stop execution of the program.

The microprocessor always begins by executing the command at location 00. It executes the commands in sequence until it reaches the Stop command.

The examples below show partial programs and describe their affect.

Program	Description
01A8	Load accumulator A with the contents of memory location 1A (26 in decimal) and stop.
01A512F8	Load accumulator A with the contents of memory location 1A (26 in decimal), decrement it, store the result to memory location 2F, then stop.

Standard Input

The input will consist of several lines of exactly 256 hex characters. Each line is the contents of memory, beginning with address 00 and ending with address FF. The end of the input is indicated by a memory state that has a stop instruction (an “8”) at address 00. The input programs will never “fall of the end of memory”, that is, you will never execute an instruction that is located between addresses F0 and FF, inclusive.

Standard Output

For each memory state, you should simulate execution beginning with address 00. When the stop instruction is reached, you will dump the contents of memory to the output as a single string of 256 hex characters followed by a newline character.

Examples

This program reads two one-word numbers from 10 and 11, and stores the two-word sum at 12 and 13.

```
0102011311321128FF0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
```

The correct output is:

```
0102011311321128FF1E00000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
```

This program computes n^2, where n is stored at 40, and stores the two-word result at FE and FF.

```
040563B14004220FF31FF041320FE31FE00C2042314200032041314170080000F03000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000001
```

The correct output is:

```
040563B14004220FF31FF041320FE31FE00C204231420003204131417008000011F000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000E1
```

Problem G : Go

(white balloon)

A *cyclic number* is an integer *n* digits in length which, when multiplied by any integer from 1 to *n*, yields a "cycle" of the digits of the original number. That is, if you consider the number after the last digit to "wrap around" back to the first digit, the sequence of digits in both numbers will be the same, though they may start at different positions.

For example, the number 142857 is cyclic, as illustrated by the following table:

142857	×	1	=	142857
142857	×	2	=	285714
142857	×	3	=	428571
142857	×	4	=	571428
142857	×	5	=	714285
142857	×	6	=	857142

Write a program which will determine whether or not numbers are cyclic.

Standard Input

The input starts with a single integer N on the first line, which is the number of the cases that will follow. This integer is followed by a list of integers from 2 to 60 digits in length, each on a single line. (Note that preceding zeros should *not* be removed, they are considered part of the number and count in determining *n*. Thus, "01" is a two-digit number, distinct from "1" which is a one-digit number.)

Standard Output

For each input integer, write a line in the output indicating whether or not it is cyclic.

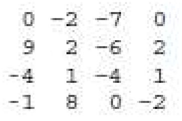
Sample Input	Sample Output
`5`	`142857 is cyclic`
`142857`	`142856 is not cyclic`
`142856`	`142858 is not cyclic`
`142858`	`01 is not cyclic`
`01`	`0588235294117647 is cyclic`
`0588235294117647`	

Problem H : Max

(black balloon)

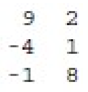
Given a two-dimensional array of positive and negative integers, a *sub-rectangle* is any contiguous sub-array of size 1 × 1 or greater located within the whole array. The sum of a rectangle is the sum of all the elements in that rectangle. In this problem the sub-rectangle with the largest sum is referred to as the *maximal sub-rectangle.*

As an example, the maximal sub-rectangle of the array:



```
 0 -2 -7  0
 9  2 -6  2
-4  1 -4  1
-1  8  0 -2
```

is in the lower left corner:



```
 9  2
-4  1
-1  8
```

and has a sum of 15.

Standard Input

The input consists of an $N \times N$ array of integers. The input begins with a single positive integer N on a line by itself, indicating the size of the square two-dimensional array. This is followed by N^2 integers separated by whitespace (spaces and newlines). These are the N^2 integers of the array, presented in row-major order. That is, all numbers in the first row, left to right, then all numbers in the second row, left to right, etc. N may be as large as 100. The numbers in the array will be in the range [-127,127].

Standard Output

Output the sum of the maximal sub-rectangle.

Sample Input	Sample Output

`4`
`0 -2 -7 0 9 2 -6 2`
`-4 1 -4 1 -1`
 	
`8 0 -2`
	`15`

Problem I : Set

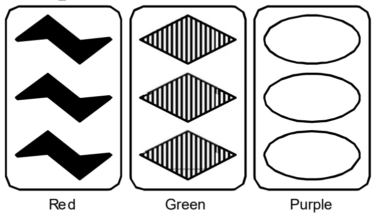
(maroon balloon)

The game of Set is played with a deck of eighty-one cards, each having the following four characteristics:

- Symbol: diamonds, ovals, or squiggles
- Count: 1, 2, or 3 symbols
- Color: red, green, or purple
- Shading: outlined, filled, or striped

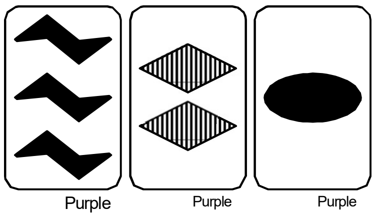
The cards are shuffled and a tableau of twelve cards is laid out. Players then attempt to be the first to identify "sets" which exist in the tableau. Sets are removed as they are identified and new cards are dealt in their place. Play continues in this manner until all cards have been used. The winner is the player with the most sets.

A *set* is a collection of three cards in which each characteristic is either the same on all three cards or different on all three cards. For example, the cards shown below form a set.



To see how the cards above form a set, take each characteristic in turn. First, each card has *different symbol*: the first card has squiggles, the second diamonds, and the third ovals. Second, each card has the *same count* of symbols, three. Third each card has a *different color*, and finally, each card has a *different shading.* Thus, each characteristic is either the same on all three cards or different on all three cards, satisfying the requirement for a set.

Consider the following example of three cards which do *not* form a set.



Again, take each characteristic in turn. Each card has a different symbol, each card has a different count of symbols, and each card is the same color. So far this satisfies the requirements for a set. When the shading characteristic is considered, however, two cards are filled and one card is striped. Thus, the shading on all three cards is neither all the same nor all different, and so these cards do not form a set.

Standard Input

The input for this program consists of several tableaus of cards. The tableaus are listed in the standard input one card per line, with a single blank line between tableaus. The end of the standard input is marked by the asterisk character on a single line, after the last tableau. Each card in a tableau is specified by four consecutive characters on the input line. The first identifies the type of symbol on the card, and will be either a "D", "O", or "S", for Diamond, Oval, or Squiggle, respectively. The second character will be the digit 1, 2, or 3, identifying the number of symbols on the card. The third identifies the color and will be an "R", "G", or "P" for Red, Green, or Purple, respectively. The final character identifies the shading and will be an "O", "F", or "S" for Outlined, Filled, or Striped. All characters will be in uppercase.

Standard Output

The output for the program is, for each tableau, a list of all possible sets which could be formed using cards in the tableau. The order in which the sets are output is not important, but your output should adhere to the format illustrated by the example below. In the event that no sets exist in a tableau, report "*** None Found ***".

Sample Input

```
S1PS
D3PO
S2GF
O2GS
O2GF
O3PO
S2RF
S3GS
D2GS
O1GS
O1GF
S2PS

O2GF
O1PF
D2PO
D3RO
S2PO
O1GF
O1GS
D2GO
S3PF
S2GF
D2GS
S1RS
*
```

Sample Output

```
CARDS: S1PS D3PO S2GF O2GS O2GF O3PO S2RF S3GS D2GS O1GS O1GF S2PS
SETS:  1.   D3PO S2RF O1GS
       2.   S3GS D2GS O1GS

CARDS: O2GF O1PF D2PO D3RO S2PO O1GF O1GS D2GO S3PF S2GF D2GS S1RS
SETS:  *** None Found ***
```