Problem name: Cipher
Input File: CipherIn.txt

In simple columnar transposition cipher, the plaintext is written horizontally onto a piece of graph paper with fixed width. The cipher text is then read vertically. For example, the sentence of "THE WEATHER IS SO NICE THAT WE WANT TO PLAY" is written as the following using a graph paper width of six:

T	H	E	W	E	A
T	H	E	R	I	S
S	O	N	I	C	E
T	H	A	T	W	E
W	A	N	T	T	O
P	L	A	Y		

The resulting output in this case would be:
TTSTWPHHOHALEENANAWRITTYEICWTASEEO

Given a string and a positive integer width, output the cipher text.

Inputs:
The number of columns to encipher the text over, followed by the phrase to encode (including spaces between each word).

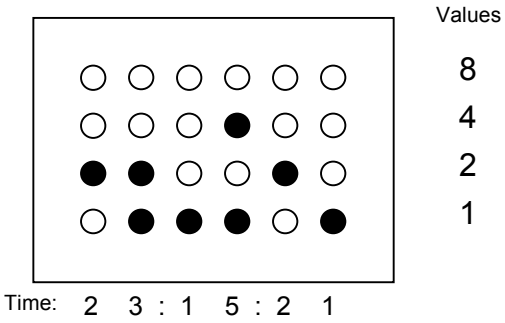
Outputs:
The enciphered phrase.

Sample Input:
6
THE WEATHER IS SO NICE THAT WE WANT TO PLAY

Sample Output:
TTSTWPHHOHALEENANAWRITTYEICWTASEEO

Problem name: Clock
Input File: ClockIn.txt

You are an employee of a company that designs and manufactures technology-related novelties. Their newest product is a small clock that displays the current time using a system of LED lights to represent the time as six binary coded decimals. The clock displays time in military (24-hour clock) format, showing (from left to right) hours, minutes, and seconds.



Military time adds 12 hours to all afternoon times-of-day from 1:00:00 pm on. Thus, the time-of-day displayed on the clock depicted above is 11:15:21 pm. Your task, should you decide to accept it, is to determine which lights must be lit on the clock in order to display the military time of day.

Inputs:
The first line of input will contain the number of military times to be displayed, followed by the military times, one per line.

Outputs:
There will be 5 lines of output for each military time to be displayed with a blank line between set of 5 lines. The first line in the group will be the military time using base 10 digits followed by the LED pattern for the military time. A 0 will indicate the LED is off, 1 one will indicate the LED is on. Each digits output on a line will be separated by 1 space.

Sample Input:
```
2
23:15:21
06:31:52
```

Sample Output:
```
2 3 1 5 2 1
0 0 0 0 0 0
0 0 0 1 0 0
1 1 0 0 1 0
0 1 1 1 0 1

0 6 3 1 5 2
0 0 0 0 0 0
```

0 1 0 0 1 0
0 1 1 0 0 1
0 0 1 1 1 0

Problem name: Credit
Input File: CreditIn.txt

Although Zack's on-line store sales are booming, he's very unhappy. Many of the purchases are being billed to invalid credit cards. As a result, the more items Zack sells the further he goes into debt. Desperate, Zack has decided to hire you and your team to develop a credit card validation program.

You have decided to use a checksum scheme to validate the card numbers. Under this scheme credit card numbers must be 16 digits in length. Since actual credit card numbers varying in length from 13 digits to 16 digits, numbers with less than 16 digits will be zero-filled from the left to create a sixteen digit number. Then, starting from the left most digit, each digit of the credit card is multiplied by a weighting factor alternating between 2 and 1, beginning with a weighting factor of 2. If the number multiplied by the weighting factor results in a 2-digit number, each digit is added to the sum. If the final sum (the checksum) is a multiple of 10, the card is valid. Otherwise, it is invalid.

Example using the *invalid* card number: 99 0011 0012 0034

0	0	9	9	0	0	1	1	0	0	1	2	0	0	3	4	
2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	
0	+0	+1+8	+9	+0	+0	+2	+1	+0	+0	+2	+2	+0	+0	+6	+4	=35

Inputs:
A credit card number varying in length from 13 to 16 digits with a space between 4 digit groupings.

Outputs:
The zero-filled (if less than 16 digits long) credit card number, the checksum, and a status of "Valid" or "Invalid" depending on the success or failure of the checksum.

Sample Input
99 0011 0012 0034
5499 0011 0012 0034

Sample Output
Card Number: 5499 0011 0012 0034
Checksum: 35
Status: Valid

Card Number: 0099 0011 0012 0034
Checksum: 40
Status: Invalid

Problem name: Inventory
Input File: InventoryIn.txt

Your favorite uncle Geoff, who owns a popular online electronics store, is about to hold his annual clearance sale. His accountant has advised him to place an item on sale if it is over-stocked *or* it is over-valued. An item is considered over-stocked if it constitutes more than 10% percent of the number of items in the inventory. It is considered over-valued if it represents more than 5% of the dollar value of the inventory. Items that are over-stocked *or* over-valued are marked down by 25%. Items that are over-stocked *and* over-valued are marked down by 50%.

Uncle Geoff has asked you to write a program to determine which items will be included in the annual clearance sale. Since he's your favorite uncle, you've agreed. He only has shelf space for 100 different types of items.

Inputs:
There will be one line of input per inventory item. Each line will contain four fields: the item's unit price, followed by the manufacturer, quantity in stock, and finally the item's name.

Outputs:
One line of output per inventory item formatted and annotated exactly as shown below.

Sample Input:
$75.95 Abit 200 Motherboard
$299.99 Canon 1 Camera
$144.33 Intel 23 Processor
$25.98 Logitech 30 Mouse

Sample Output:
Abit Motherboard 75.95 200 **** Sale item, price: 37.98 ****
Canon Camera 299.99 1
Intel Processor 144.33 23
Logitech Mouse 25.98 30 **** Sale item, price: 19.48 ****

Problem name: Roots
Input File: RootsIn.txt

The square root of a number, y, can be found using Newton's Approximation Formula. In this approximation technique x_0, x_1, x_2, … are a sequence of successive approximations of the square root of y defined as follows:

x_0 = a seed value
for all $i > 0$, $x_{i+1} = (1/2) * ((y / x_i) + x_i)$

Both y and x_0 must be positive.

You are to demonstrate that Newton's Approximation Formula converges on a given number's square root by outputting the successive approximations until the approximation is within a given tolerance of the number's square root.

Inputs:

The first line will contain the tolerance to be used to terminate the successive approximations followed by the seed value. The next line will contain the number of values whose square root is to be calculated followed by the values themselves, one per line.

Outputs:

There will be one output grouping per calculated square root. Each grouping will contain the successive approximations of the square root of the given number, beginning with x_0, one approximation per line until the calculated approximation is within the given tolerance of the actual square root of the number. Each line in the grouping will contain the value of the approximation, the actual square root of the number, and finally the difference between the calculated and actual square root of the number. Three digits will be displayed to the right of the decimal point. Groupings will be separated by a blank line.

Sample Input:

```
0.05 1234.0
2
49.34
10765.42
```

Sample Output:

```
1234.000        7.024        1226.976
617.020         7.024        609.996
308.550         7.024        301.526
154.355         7.024        147.331
77.337          7.024        70.313
38.988          7.024        31.963
20.127          7.024        13.102
11.289          7.024        4.265
7.830           7.024        0.806
```

7.066	7.024	0.041
1234.000	103.757	1130.243
621.362	103.757	517.605
319.344	103.757	215.587
176.527	103.757	72.771
118.756	103.757	14.999
104.704	103.757	0.947
103.761	103.757	0.004

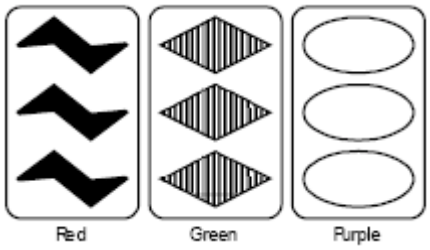
Problem name: Sets
Input File: SetsIn.txt

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

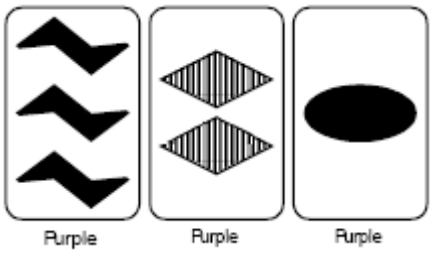
The cards are shuffled and a tableau of twelve cards is laid out. Players then attempt to be the first to identify "sets" which exist in the tableau. Sets are removed as they are identified and new cards are dealt in their place. Play continues in this manner until all cards have been used. The winner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

Inputs:

The input for this program consists of several tableaus of cards. The tableaus are listed in the input file one card per line, with a single blank line between tableaus. The end of the input is marked by the end of the file. 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 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.

Outputs:

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

Problem name: Tolls
Input File: ToolsIn.txt

Each square of a 10x10 checkerboard has a toll associated with it that must be paid when you enter the square. You wish to travel from the bottom most row to the top most row and minimize the total of the tolls along the way. Write a program to output the row and column numbers of a route that minimizes the tolls. When making a move, the row number *must* increase by 1 and the column number can change by -1, 0, or +1. Tolls range from 0 to 9, and row 1 and column 1 is the lower left most square of the checkerboard.

	column 1	2	3	4	5	6	7	8	9	10
10	6	4	7	4	8	3	6	7	2	4
9	9	1	4	7	3	6	8	6	1	4
8	4	8	1	9	7	9	2	3	5	4
7	1	8	6	6	8	4	8	3	8	2
6	7	3	7	4	4	1	5	9	9	4
5	1	6	3	2	1	4	3	3	7	9
4	5	3	8	4	2	6	7	9	3	5
3	6	4	3	8	7	1	2	4	7	4
2	8	8	3	6	5	8	3	9	1	5
row 1	0	3	5	6	1	2	7	1	9	4

Inputs:

The tolls associated with each square of the checkerboard, one line per row. The first line of input is the tolls associated with row 10, the second line the tolls associated with row 9, etc. The tolls on a line will be separated by a space.

Outputs:

The minimum total toll followed by 10 lines that give the row and column numbers of the minimum toll path through the checkerboard. Each square's location will be on a separate line, with the row number preceding the column number. There will be a space between each row and column number.

Sample Input:

```
6 4 7 4 8 3 6 7 2 4
9 1 4 7 3 6 8 6 1 4
4 8 1 9 7 9 2 3 5 4
1 8 6 6 8 4 8 3 8 2
7 3 7 4 4 1 5 9 9 4
1 6 3 2 1 4 3 3 7 9
5 3 8 4 2 6 7 9 3 5
6 4 3 8 7 1 2 4 7 4
8 8 3 6 5 8 3 9 1 5
0 3 5 6 1 2 7 1 9 4
```

Sample Output:

```
23
1 8
2 7
3 6
4 5
5 5
6 6
7 7
8 8
9 9
10 9
```