Problem 1: Nbits

How many different integers between A and B (including A and B) have exactly N bits of 1 in the two's complement representation? That's the question to be answered in this problem.

Example

The problem is pretty clear. For example, suppose A is 5, B is 14, and N is 2. If we look at the two's complement binary representation of the integers between 5 and 14 and identify those with exactly 2 one bits, we find that there are five such numbers (identified by the left-pointing arrows)[1]:

5	0 1 0 1 ←	10	1 0 1 0 ←
6	0 1 1 0 ←	11	1 0 1 1
7	0 1 1 1	12	1 1 0 0 ←
8	1 0 0 0	13	1 1 0 1
9	1 0 0 1 ←	14	1 1 1 0

So the answer for this case would be 5.

Input

There will be multiple input cases to consider. For each case there will be a single input line containing A, B, and N. The input for the last case will be followed by a line containing three zeroes. A and B will each be in the range -2147483648 to +2147483647, and N will be in the range 1 to 32.

Output

For each input case, display the case number (1, 2, …) and the appropriate number. Display a blank line after the output for each case. The sample input and output illustrate the appropriate formats.

Sample Input	Output for the Sample Input
5 14 2 5 14 3 -1 1 1 0 0 0	Case 1: 5 numbers Case 2: 4 numbers Case 3: 1 numbers

[1] All the high-order bits in these numbers are 0; they are not shown for clarity.

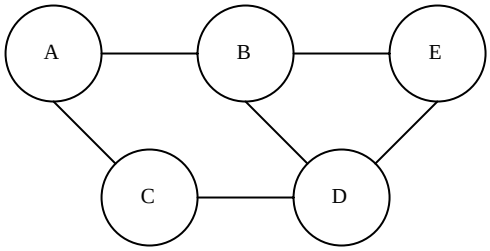
Problem 2: Bandwidth

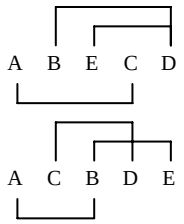
Given an undirected graph (*V*, *E*), where *V* is a set of vertices and *E* is a set of edges, and an ordering on the elements in V, then the *bandwidth* of a vertex *v* is defined as the maximum distance in the ordering between *v* and any node to which it is directly connected in the graph, or 0 if v is not connected to any other vertices. The bandwidth of an ordering is defined as the maximum of the bandwidths of the vertices in the ordering.

The distance between two vertices A and B in an ordering is just one more than the number of vertices between A and B in the ordering. So if an ordering includes … A P Q R B …, then the distance between A and B in that ordering is 4 (assuming A is directly connected to B in the graph).

Example

For example, consider the graph shown below on the left.





The vertices in this graph may be ordered in many different ways, two of which are illustrated on the right. For the first ordering (A, B, E, C, D), the bandwidths of the vertices A, B, E, C and D are 3, 3, 2, 3, and 3; the bandwidth of the ordering is thus 3, the maximum vertex bandwidth. The second ordering (A, C, B, D, E) has vertex bandwidths of 2, 2, 2, 2, and 2, which yields an ordering bandwidth of 2.

The Problem

Your task is to write a program that will determine the ordering of the vertices in a graph that will yield the *minimum* bandwidth.

Input

There will be multiple input cases to consider. For each case there will be a single input line containing a description of a graph. A line will consist of a series of vertex descriptions separated from each other by semicolons. Each vertex description will consist of a vertex name (a single uppercase letter in the range 'A' to 'Z') followed by a colon and the name of one (or more) of the vertices to which it is directly connected. The input line for the last case will be followed by a line containing only $ in the first column. The graph will contain no more than 10 vertices.

Output

For each input case, display the case number (1, 2, …), the names of the vertices (separated from each other by a space), and the minimal bandwidth (in parentheses). If more than one ordering produces the same minimal bandwidth, then display the one that has the smallest lexicographic ordering. Display a blank line after the output for each case.

Sample Input	Output for the Sample Input
A:BC;B:DE;E:D;C:D A:DW;B:AES;S:TD;D:E;S:E;T:W $	Case 1: A C B D E (2) Case 2: B E A S D W T (3)

Problem 3: Big Roots

Given an integer n, $1 \le n \le 200$, and an integer p, $1 \le p \le 10^{101}$, you are to write a program that determines the positive integer value k such that $k = \sqrt[n]{p}$, if such a value exists. If such a positive integer k does not exist, your program must report that fact. You may assume the value of k will fit in a 32-bit signed integer.

Input

There will be multiple input cases to consider. For each case there will be two input lines. The first line will contain the value of n, and the second line will contain the value of p. The line following the input for the last case will contain a single integer 0.

Output

For each input case, display the case number (1, 2, …) and the value of k or the phrase "No solution", as appropriate. Display a blank line after the output for each case. The sample input and output illustrate the appropriate formats.

Sample Input	Output for the Sample Input
2 16 3 27 7 4357186184021382204544 7 4357186184021382204545 0	Case 1: 4 Case 2: 3 Case 3: 1234 Case 4: No solution

Problem 4: Zombies!

In UNIX®-like operating systems, the *fork* system call creates a new process (an independent executable object) that is uniquely identified by an integer "process ID". This new process is a "child" of its parent process (the process that executed the *fork* system call). Of course, a child process could itself execute *fork* and create its own children; there is no conceptual limit to the number of generations of processes that may exist, nor is there a limit on the number of child processes a parent may have (at least for this problem).

Another system call, *wait*[1], allows a parent process to delay its execution (if necessary) until it receives notice that one of its child processes has terminated. If a child process terminates before its parent calls *wait*, then the child becomes a so-called *zombie* process – it continues to exist but does not actually execute; if the parent later calls *wait* and at least one of its children has become a zombie, then the parent continues execution and the child process becomes completely terminated. If the parent should terminate before waiting for the termination of one or more of its children, then those remaining children become "wards of the state" – they are effectively adopted by a system process that spends it life repeatedly executing the *wait* system call, efficiently "reaping" terminated child processes that were abandoned by their parents. If a parent calls *wait* and there are no child processes, the *wait* call is effectively ignored and the parent continues execution. In this problem, a process terminates (or perhaps becomes a zombie) by executing the *exit* system call. What we seek to find in this problem is the state of all processes after a sequence of fork, exit, and wait system calls has been executed.

We assume that there is initially only a single process, with a process ID of 1, running at the beginning of each case. It's "parent" is the operating system, so when it exits, it is immediately "reaped" by the system.

If there are multiple zombie children from which a parent could select when it executes *wait*, then it selects the one with the smallest process ID.

Any zombie children of a parent are "reaped" (that is, completely terminated) as soon as the parent executes the *exit* system call.

When a process forks, the child process it creates is assigned the smallest positive process ID that has never been used. Thus the first process created in each input case will have ID 2.

Input

There will be multiple input cases to consider. For each case there will be one or more lines of input. Each line contains a positive integer giving a process ID (which will be no larger than 9999), a space, one of the words "fork", "wait", or "exit" (always using lower-case letters), and the end of line. A line containing only 0 (zero) and the end of line follows the last input line for each case. A line with the same content (0 and end of line) immediately follows the input for the last case.

There will be no input in which a process "exits" after a "wait" if the wait did not complete (either successfully or unsuccessfully). That is, there will be no illogical sequences of actions.

In those cases where all processes terminate, the end of input immediately follows the exit which terminates the last process.

[1] In real systems there is a family of wait-like system calls; we consider only a single simple wait system call in this problem.

Output

For each case, display the case number (1, 2, …), and for each state (running, zombie, and waiting) the number of processes in the state, the state name, a colon, and the process IDs (in ascending order) of processes in that state, after all input actions for the case have been taken. If no processes are left running, then state that explicitly. Display a blank line after the output for each case. Your output should be essentially identical to that shown in the sample.

Sample Input	Output for the Sample Input

1 exit
0
1 fork
2 exit
0
1 fork
2 exit
1 wait
0
1 fork
1 wait
2 exit
0
1 fork
2 fork
2 fork
2 exit
4 exit
0
0
	Case 1:
	  No processes.
	
	Case 2:
	  1 Running: 1
	  1 Zombie: 2
	  0 Waiting:
	
	Case 3:
	  1 Running: 1
	  0 Zombie:
	  0 Waiting:
	
	Case 4:
	  1 Running: 1
	  0 Zombie:
	  0 Waiting:
	
	Case 5:
	  2 Running: 1, 3
	  1 Zombie: 2
	  0 Waiting:

Problem 5: Semigroups

A binary operation on a set *S* is a mapping of every ordered pair of elements from *S* to another element of *S*. Addition of integers is such an operation, since the sum of every pair of integers is another integer. But division of integers is not a binary operation, since the result of some division operations is not an integer.

The word *ordered* is important in the definition of a binary operation, since changing the order of the operands may well yield a different result. For example, in integer subtraction, the result of 1 – 2 is certainly not the same as 2 – 1. Binary operations for which the order of the operands is unimportant are called *commutative*. That is, if for some operator @ we have a @ b = b @ a for all possible values of a and b, then the operator @ is commutative.

The examples used above, integer addition and subtraction, are defined on infinite sets. But many operations are defined on sets with a finite number of elements. For example, logical operations like and, or, and exclusive or are defined on the set of logical values, false and true. In this problem we are concerned only with operators on finite sets.

A convenient way of documenting an operator is to enumerate the elements of the (finite) set on which it is defined, and to display a table showing the result of the operator. Each row and each column in this table is labeled with a set element, and the result of the operator is given in the body of the table.

For example, consider the logical and operator. The set of elements is, of course { false, true }, and the table might be displayed as follows:

	false	true
false	false	false
true	false	true

A binary operator @ is said to be *associative* if, for all possible operands a, b, c we have (a @ b) @ c = a @ (b @ c). As we can see, the logical and operator is associative.

If a binary operation @ on a set S is associative, then the pair (S, @) forms a semigroup. If the binary operation is also commutative, then the semigroup is also said to be commutative.

The Problem

In this problem you will be given a description of an operator – that is, the elements of the finite set on which the operation is defined, and the body of the table defining the result of the operator on every pair of operands. Your program must then identify the first of these phrases that applies:

is not a binary operation.
is not a semigroup.
is not a commutative semigroup.
is a commutative semigroup.

The first statement applies if the result of applying the operator to at least one pair of operands yields a result that is not an element of the finite set on which the operator is defined.

Input

There will be multiple cases to consider. The first line of the input for each case will consist of the elements of the set on which the operator is defined. These will be given as N unique lowercase alphabetic characters; N will be no larger than 10. The next N lines of the input give the body of the table describing the result of applying the binary operator to two operands. Each line will contain N lowercase alphabetic characters. The rows and columns of the table are given in the same order as the elements on the first line of input for the case. The last case is followed by a line containing only an end of line character. There will be no spaces or tab characters in the input.

Output

For each input case, the output is the case number and one of the phrases specified above. The sample input and output illustrates the appropriate format. Display a blank line following the output for each case.

Sample Input	Output for the Sample Input

ft
ff
ft
xy
yx
xz
abc
abc
bca
cba
ab
aa
bb
(end of line only)
	`Case 1: The operator is a commutative semigroup.`
	
	`Case 2: The operator is not a binary operation.`
	
	`Case 3: The operator is not a semigroup.`
	
	`Case 4: The operator is not a commutative semigroup.`

Problem 6: The Last Term

An integer f is a positive factor of an integer d if f is greater than zero and there exists some integer n such that f × n = d. Thus 12 is a factor of 60 because 12 × 5 = 60.

A sequence of integers $x_1, x_2, \ldots, x_n$ is a decimal-digit factor sequence (DDF) if each x_i is a positive integer, $x_1 > 1$, and x_{i+1} (for all $i \geq 1$) is the sum of the digits of all positive factors of x_i.

Example

The numbers 17, 9, 13, 5, 6, … form a DDF as we can see from the following observations.

- The positive factors of 17 are 1 and 17, and 1 + 1 + 7 = 9.
- The positive factors of 9 are 1, 3, and 9; 1 + 3 + 9 = 13.
- The positive factors of 13 are 1 and 13; 1 + 1 + 3 = 5.
- The positive factors of 5 are 1 and 5; 1 + 5 = 6.

Every DDF beginning with a number greater than or equal to 1000 repeats no number greater than or equal to 1000, and also contains a number less than 1000. Also, every DDF beginning with a number less than 1000 contains no number greater than 999. Thus every DDF must eventually repeat numbers less than 1000. It has also been shown that every DDF eventually repeats a single number x_n which is called the *last term*. That is, there exists x_n such that for all $j > n$, $x_j = x_n$.

In this problem you are to determine the length of DDFs that begin with a given value. The length of a DDF is the value of n, where x_n is the last term of the DDF.

Input

There will be multiple input cases to consider. For each case there will be a single input line containing the first term x_1 of the DDF to be considered. This value will never be larger than 2000. The input for the last case will be followed by a line containing 0.

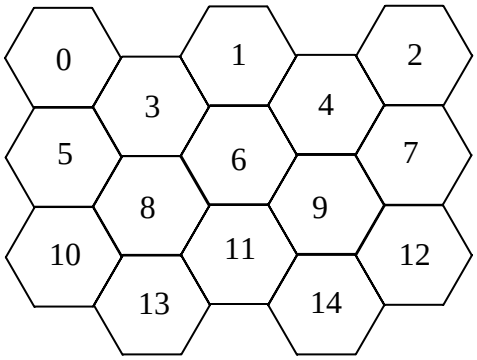
Output

For each input case, display the case number (1, 2, …) and the length of the DDF. Display a blank line after the output for each case. The sample input and output illustrate the appropriate formats.

Sample Input	Output for the Sample Input
17 31 68 1448 0	Case 1: 13 terms Case 2: 11 terms Case 3: 19 terms Case 4: 20 terms

Problem 7: Beevaria

An eastern European country called Beevaria is setting up a new jail. It is an unusual jail in that all of the cells are made with walls that are clear and unbreakable. The government wants to cut costs and, as such, has selected an unusual jail cell arrangement: each cell is a hexagon. The government has contracted with a construction firm to make these jails in various parts of the country using standardized materials. All that the construction firm needs to know is the size of the building, which the government specifies using the number of cells across the top.



Consider the figure at the right. The size of this jail is S=3, consisting of 3 alternating pairs of rows starting with 3 hexagons and then 2 hexagons each, i.e. 3, 2, 3, 2, 3, 2. If the construction crew builds a jail of size S=2, the arrangement is 2, 1, 2, 1. For S=4, it is 4, 3, 4, 3, 4, 3, 4, 3. Each cell also has a number, as shown in the diagram. The numbering scheme matches the value for S, of course, so that for S=5 the first row contains 0..4, the second contains 5..8, etc.

Beevaria never has many criminals; jails are rarely full. In fact, a prison is never so full that the number of prisoners plus guards exceeds the number of cells. In the interests of saving money, the Beevaria government has opted to only supply a sufficient number of guards as needed to watch all of the prisoners. Since the walls are clear, this makes sense; if there is a prisoner in cell 13 and another in cell 14, only one guard is needed as long as the guard sits in cell 11. That guard can see both prisoners at the same time. Guards can only see into cells that are along a line of sight perpendicular to any of the six glass walls of the cell where they are located. For instance the guard in cell 11 can watch 1, 5, 6, 7, 8, 9, 13, and 14 but cannot watch 0, 2, 3, 4, 10 or 12.

Both a guard and a prisoner cannot occupy a cell: these prisoners may be dangerous criminals!

Examples

Here are some combinations to think about, using the S=3 diagram from above:

Prisoners:	Guard position(s):	
0, 1, 2	6	
1, 4, 5	6, 11	(for example, also others)
9, 11, 13	7	(note that the guard can see all three)
5, 6, 12, 13	3	
0, 5, 9, 10, 12	3, 8	(for example, also 3, 4, and others)
2, 4, 6, 8, 10	1, 5, 7	(for example)

The problem

When presented with a value for prison size S and a set of cell numbers for prisoners, determine the *minimum number* of guards that are needed to keep watch over all of the prisoners.

Input

There will be multiple prison descriptions in the input. Each description gives the prison size S and the numbers of the cells housing prisoners. The input is a sequence of integers, one per line. The

integer on the first line of a description specifies the size S of a prison, which is never larger than 25. This is followed by one line for each prisoner in the prison giving the prisoner's cell number. A -1 (on a line by itself) follows the cell number for the last prisoner, and a -1 (on a line by itself) follows the last prison description in the input.

Output

For each input prison description, print the prison number (1, 2, …) and the number of guards needed to watch the prisoners in the format shown below. Display a blank line after the output for each prison.

Sample Input	**Output for the Sample Input**
3 **0** **5** **9** **10** **12** **-1** **3** **2** **4** **6** **8** **10** **-1** **-1**	Prison 1, Guards needed: 2 Prison 2, Guards needed: 3

Problem 8: Roman Palindromes

No wonder Rome fell. Their numbering system was just messed up. Consider the number 4, which in Roman Numerals is "IV". If I reverse the letters and make "VI" I get 6. And in general, for any Roman numeral letters A_1 and A_2, if A_1's value is < A_2's value, then $A_1A_2 = A_2 - A_1$. This is like "IV" = 4. But for A_2A_1 you add, like "VI" = 6.

I = 1
V = 5
X = 10
L = 50
C = 100
D = 500
M = 1000

A table of the Roman Numeral symbols and their decimal equivalents is at right.

We can generalize the A_1A_2 case and the A_2A_1 case with any of the letters. For example, "VC" = 95, but "CV" = 105, while "XM" = 990, and "MX" = 1010. These combinations can also occur "in the middle" of a Roman Numeral: "XXIX" is 29, "CDXLV" is 445.

Now suppose one were to take a Roman numeral string and generate a palindrome. Flip the original string end-to-end and "prepend" it in front of the original string according to an even/odd rule: If the length of the original string is even, the first character appears twice, but if the length is odd the first character appears only once. Thus the string "MDC" becomes "CDMDC", and the string "VI" becomes "IVVI".

Apply the following rules and consider A_1=string[i] and A_2=string[i+1] for i=1..length-1*:

1. For letters A_1 and A_2, if A_1's value is < A_2's value, then $A_1A_2 = A_2-A_1$
2. Else if the letters A_1 and A_2 are $A_1 \geq A_2$, then $A_1A_2 = A_1+A_2$

The result of the Roman Palindrome is the sum of the answers for all A_1A_2.

Examples

Input	Palindrome	Value
VI	IVVI	I and V = 5 – 1 = 4 V and V = 5 + 5 = 10 V and I = 5 + 1 = 6 Total Roman Palindrome number = 20
MCM	MCMCM	M and C = 1000 + 100 = 1100 C and M = 1000 – 100 = 900 M and C = 1000 + 100 = 1100 C and M = 1000 – 100 = 900 Total = 4000
MCMLXXIV	VIXXLMCMMCMLXXIV	V and I = 5 + 1 = 6 I and X = 10 – 1 = 9 X and X = 10 + 10 = 20 X and L = 50 – 10 = 40 L and M = 1000 – 50 = 950 M and C = 1000 + 100 = 1100 C and M = 1000 – 100 = 900 M and M = 1000 + 1000 = 2000

* Not to be confusing, but if the arrays used to hold the strings are zero-based, this is i=0..length-2; we are just indicating that you consider each pair of consecutive characters in the string.

		M and C = 1000 + 100 = 1100 C and M = 1000 – 100 = 900 M and L = 1000 + 50 = 1050 L and X = 50 + 10 = 60 X and X = 10 + 10 = 20 X and I = 10 + 1 = 11 I and V = 5 – 1 = 4 Total = 8170

Given a string using characters from the set { I, V, X, L, C, D, M }, print the value of the Roman Palindrome.

Input

There will be multiple input cases to consider. Each case has a single input line containing the string of Roman numeral symbols (never more than 10) followed immediately by the end of line. The line following the input for the last case contains only an end of line.

Output

For each input case, display the case number (1, 2, …), and the value of the Roman Palindrome. Your output should be similar to that shown in the sample. Display a blank line after each line of output.

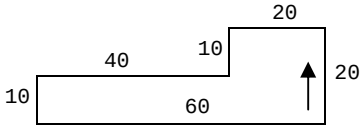
Sample Input	Output for the Sample Input
VI **MCM** **MCMLXXIV** ***(only an end of line)***	Case 1: total = 20 Case 2: total = 4000 Case 3: total = 8170

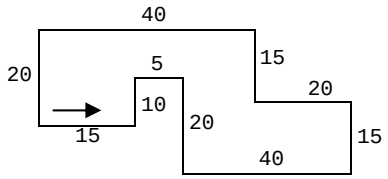
Problem 9: The Wall Walker

Frugal Floors is in the business of selling and installing carpet and hardwood floors. Part of their job is to accurately determine the amount of material needed to cover the floor in a room. Because many rooms aren't simple rectangles, Frugal Floors employs a small robot to record the outline of the floor. Assuming the room is empty (or at least has all the furniture moved away from the walls), the robot starts at the intersection of two walls. It then moves along a wall, always staying in contact with a wall, recording its path, until it returns to the starting point. Frugal Floors – as part of being frugal – always requires their customer's rooms to have square corners. Their employees are also careful to close the doors to the rooms being measured to avoid having their robot wander into other rooms!

Once the robot completes its trip around the room's periphery, the data it recorded is downloaded to a computer and used to compute various items. In particular, the area of the floor is an important value. In this problem you will write a program that will compute that area from the robot's route.

The route data is simple: it consists of a sequence of pairs (distance,direction), where "distance" is the distance the robot travels along a wall, and "direction" is left or right, indicating the way the robot turned to continue along the next wall. Let's consider a few examples.





In the room on the left, the robot begins in the lower right corner and travels 20 feet, as indicated by the arrow. It then turns left and travels another 20 feet. It turns left again and travels 10 feet. After a right turn it travels 40 feet. Another left turn, 10 feet more, another left turn, and 60 feet more bring the robot back to its starting point.

To illustrate the input data format, let's consider the robot's data for the room on the right. It will be this:

15 L 10 R 5 R 20 L 40 L 15 L 20 R 15 L 40 L 20 X

The input is an integer giving the number of distance units[1] the robot moved, a space, the letter L or R, a space, another integer distance, and so forth until the robot reaches its starting point, at which point it emits the letter X and the end of line (instead or L or R).

The problem, of course, is to determine the area of the room measured by the robot. You are assured that the robot's data is correct, and that only real rooms with positive non-zero areas are being measured. You may also safely assume the area of each room is no greater than 100,000 square feet, and that no wall is longer than 1,000 feet.

[1] We use the foot as the unit of measurement in this problem, but the real robot can be configured to use much smaller units, like centimeters, for accuracy.

Input

There will be multiple cases to consider. The input for each case is contained entirely in a single input line, and consists of an alternating sequence of integers and capital letters (L or R or X), beginning with an integer and ending with the letter X. Adjacent items are separated by exactly one space. The end of line immediately follows the X. A line containing only an X and the end of line follows the input for the last case.

Output

For each input case, display the case number (1, 2, …) and the area of the room. The sample input and output illustrates the appropriate format. Display a blank line after the output for each case.

Sample Input	Output for the Sample Input
20 L 20 L 10 R 40 L 10 L 60 X 15 L 10 R 5 R 20 L 40 L 15 L 20 R 15 L 40 L 20 X X	Case 1: 800 sq. ft. Case 2: 1250 sq. ft.

Problem 10: Trick, or Treat?

Frugal Floors (of problem 9 fame) wants to provide "frugal" floorplans to their customers showing the layout of rooms. Being frugal, they have an inexpensive printer that can only print text characters with fixed widths (that is, it uses "monospaced" fonts). It also has no facility for backspacing, doing "reverse line feeds" (that is, moving the paper backward), or overprinting. But the printer can still display a simple text-mode approximation of a floorplan.

They would like to employ exactly the same input data used in the program that calculates the area of a room (see problem 9 for details). At the intersection of each pair of walls, they want to display a plus sign ('+'). They want to display hyphens ('-') or vertical strokes ('|') for the walls, using whichever is appropriate.

For a wall of length N, they want to use 2 × N + 1 characters; two of these will be plus signs, and the remainder will be hyphens or vertical strokes. The only other characters that may "appear" in the floorplan are spaces (blanks) and end of line characters.

Consider a few examples. For a square room with 6-foot walls they want the diagram shown on the left below. And for a similar room with a 1-foot square inset at the lower right corner, they want the diagram shown on the right.

```
+-----------+
|           |
|           |
|           |
|           |
|           |
|           |
|           |
|           |
|           |
|           |
|           |
+-----------+
```

```
+-----------+
|           |
|           |
|           |
|           |
|           |
|           |
|           |
|           |
|           |
|         +-+
|         |
+---------+
```

Although these diagrams are correct, the vertical and horizontal sizes of a character cell on the printer differ, so the diagrams appear elongated. That's certainly expected.

And since the robot that prepares the data could be started at the intersection of any pair of walls, and can move clockwise or counterclockwise around the room's periphery, the diagrams may be rotations or mirror images of those shown in these examples. You may be assured that this behavior is anticipated as well, and is acceptable.

Although real printers have limitations on the widths of the lines they may display, Frugal apparently forgot this, and therefore didn't specify the scaling of floorplans that are wider than the printer. When such wide floorplans are printed, the long lines may wrap around or be truncated, but that's not your problem.

Input

The input is in exactly the same format as that used for problem 9.

Output

The output for each case is a line containing the case number (as illustrated in the samples below), followed by the text-mode floorplan starting on the next line. Display a blank line after each floorplan.

Sample Input	Output for the Sample Input
6 L 6 L 6 L 6 X 5 L 1 R 1 L 5 L 6 L 6 X X	Case 1: +----------+ | | | | | | | | | | | | | | | | | | | | +----------+ Case 2: +----------+ | | | | | | | | | | | | | | | | | | | +-+ | | +---------+