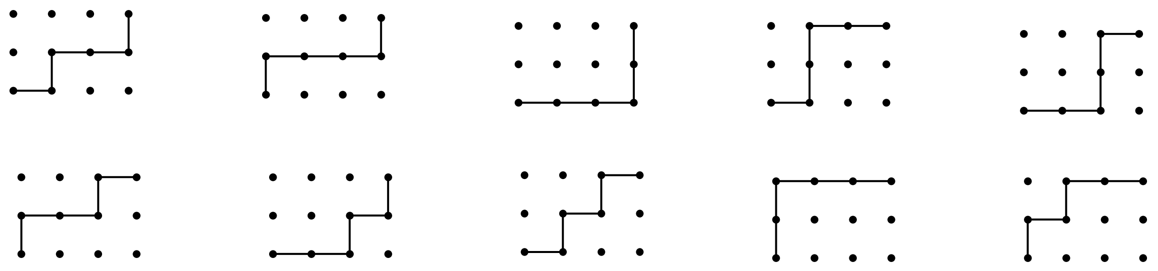
Problem 1: Jill's Walks

Jill Bates used to be an avid cyclist, but as she's grown older, she now walks for exercise. The regions where Jill walks are all laid out as rectangular grids, and she arranges her walks so she walks from the intersection at the lower left of the region (on a map) to the upper right intersection, always moving away from her starting point and closer to the destination.

For example, consider a region of size 3 by 4 with 12 intersections. Here are the 10 possible routes Jill could take through this region.



Given the dimensions of the region, how many different routes could Jill take?

Input

There will be multiple input cases to consider. The input for each case is a line containing a pair of integers M and N giving the number of intersections in each dimension of the region. Each of these is greater than zero and no larger than 50. The input for the last case is followed by a line containing two zeroes. No signed integers larger than 64 bits will be required in a solution.

Output

For each case, display the case number and number of unique paths Jill might take. Each of the answers will fit in a 32-bit signed integer.

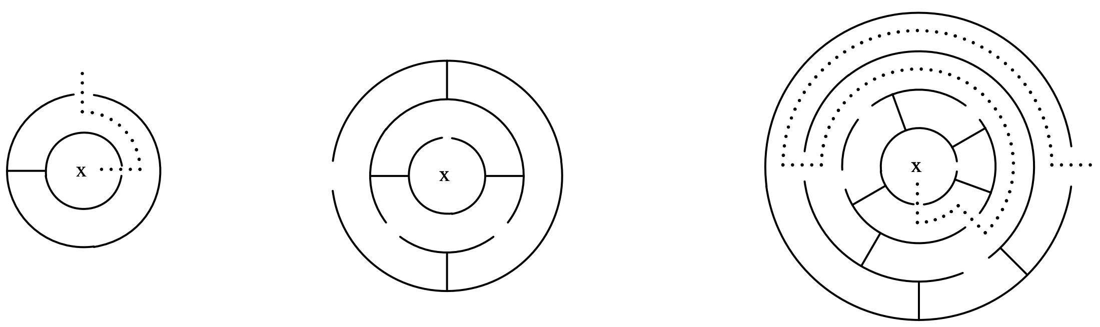
Sample Input	Output for the Sample Input

`3 4`
`4 3`
`1 50`
`2 10`
`0 0`
	`Case 1: 10`
	`Case 2: 10`
	`Case 3: 1`
	`Case 4: 10`

Problem 2: Circular Mazes

A circular maze consists of a number of concentric circles. Each circle may have *gates* that permit passage between the regions enclosed by the circles, and there may be *walls* joining adjacent pairs of concentric circles that block passage. Given the description of a circular maze, your job is to find a path from outside the maze to the innermost circle.

Here are three examples. In each case, "x" marks the spot you are to reach, and the dotted line shows the path to be taken, except for the middle maze, which has no solution. For comparison with the sample input/output, angles are measured counterclockwise from the east, which is 0 degrees.



Input

There will be multiple input cases to consider. The first line of input for each case contains an integer *N* (1 ≤ *N* ≤ 10) that specifies the number of concentric circles in the maze. This line is followed by *N* lines, each giving integer values specifying (a) the number of gates in a circle, (b) the angular position of those gates (in any order), (c) the number of walls connecting the circle to the next inner circle, and (d) the angular position of those walls (in any order). These *N* lines are ordered from the innermost circle to the outermost. The innermost circle will never have any walls, and no circle will ever have more than 10 gates or 10 walls. The line following the input for the last case will contain an integer 0.

Output

For each input case, display the case number (1, 2, …) and the path from outside the maze to the interior of the innermost circle as an alternating sequence of integer angles and the direction of movement between circles (IN to the next inner circle, or OUT to the next outer circle). Each angle and direction should be separated by a single space. Any correct solution for a case will be accepted, except those that retrace all or part of a path. Display "No solution" if there is no path.

Example

For example, consider the leftmost maze shown above. The first three lines of the sample input (shown on the next page) correspond to this maze. The first line (`2`) indicates there are two concentric circles in the maze. The second line (`1 0 0`), corresponding to the innermost circle, indicates 1 gate located at 0 degrees, and no (0) walls. The third line (`1 90 1 180`), corresponding to the next (and outermost) circle, indicates 1 gate located at 90 degrees, and 1 wall located at 180 degrees.

The solution, as illustrated by the dotted line, indicates entry to the maze through the outermost circle at 90 degrees ("north") and entry to the innermost circle at 0 degrees ("east"). This is shown in the solution as `90 IN 0 IN`. Although `90 IN 90 OUT 90 IN 0 IN` would logically still be a valid solution, repetition of all or part of a path is not permitted.

Sample Input

```
2
1 0              0
1 90             1 180
3
1 90             0
2 225 315        2 0 180
1 180            2 90 270
4
2 0 270          0
4 45 135 190 315 4 30 210 340 110
2 180 300        1 240
1 0              2 270 315
0
```

Output for the Sample Input

```
Case 1: 90 IN 0 IN
Case 2: No solution
Case 3: 0 IN 180 IN 315 IN 270 IN
```

Problem 3: Dice Game

A certain game involves rolling five dice and scoring according to a score sheet. At each turn, the player rolls all five dice. Next, the player considers the roll and potentially removes and re-rolls some or all of the dice with the intention of potentially improving the score. This happens again, and after two re-rolls the final dice are used for scoring.

The score sheet is divided into two halves, the top half and (wait for it!) the bottom half. The bottom half is organized like various poker hands, such as three-of-a-kind and a full-house. The problem here only concerns the top half of the score sheet.

In the top half of the sheet, the player takes the final roll and adds up the dice according to whether they want "ones", "twos", and so on. In other words, if the five dice show 3-4-3-5-2, the player could opt to fill in the "threes" slot, and then adds the two occurrences of threes that are showing, for a total score of six. As another example, if a player were to roll 2-2-3-2-6, the score could be recorded as six in the twos slot. The player must fill in each of the slots: ones, twos, … up to the sixes slot. When all of the slots are filled, the total for the upper half is the sum of the scores in each slot. If a player scores a total of at least 63 points in the upper half, a bonus of 35 points is added to the upper section score. This 63 points corresponds to three-of-a-kind for each of the six dice faces. A common way to get the bonus is rolling four or five of a larger number so that fewer of the smaller numbers are needed (a player can earn the bonus even if he or she scores a "0" in an upper section slot).

Since there are six upper half slots to fill, it will require six rolls of the dice. When actually playing the game, one of the variables is when to fill in each slot, and once an entry is placed on the score sheet it remains for the remainder of the game. So a player rolling 6-6-3-6-2 should probably score this as an 18 in the six slot, not a three in the three slot or a two in the two slot. In our case we have advance knowledge of all dice rolls, so we can use this to maximize the potential score and place the rolls of the dice into the correct slots.

Example

Consider these dice rolls:

2-3-2-6-1   1-3-1-1-1   4-3-4-6-4   2-3-5-5-6   6-3-5-3-5   1-4-4-6-1

There are a number of ways that these rolls can be used to fill the slots on the upper half of the score sheet. Here are two of them:

Roll					Score		Roll					Score
1	3	1	1	1	4	ones	1	4	4	6	1	2
2	3	2	6	1	4	twos	2	3	2	6	1	4
6	3	5	3	5	6	threes	1	3	1	1	1	3
4	3	4	6	4	12	fours	2	3	5	5	6	0
2	3	5	5	6	10	fives	6	3	5	3	5	10
1	4	4	6	1	6	sixes	4	3	4	6	4	6
Sub Total					42		Sub Total					25
35 point bonus for >= 63?					0		35 point bonus for >= 63?					0
Total					42		Total					25

Obviously there are other potential orderings and totals, and that at least based on the above, a score of 42 is better than a score of 25. Remember that if the sub-total is at least 63, then there is a 35 point bonus, although this does not occur in this example.

The problem

Write a program to accept one or more sets of six rolls of five dice, and to display the maximum number of points that can be scored given the above rules and the given set of dice rolls. All that is necessary is to display this maximum score; it is not necessary to explain the placing of the rolls into the slots.

Input

There will be multiple sets of input. Each set contains six lines of data in the format described above: five single-digit numbers with dashes between each. Thus each line contains exactly 9 characters followed by the end of the line. For each set of six lines, display the maximum possible score. There will be a single line of all zeros ("0-0-0-0-0") after the last set of rolls.

Output

For each input data set display the input set case number (1, 2, …) and the maximum possible score using the input rolls. Use the format shown in the sample output.

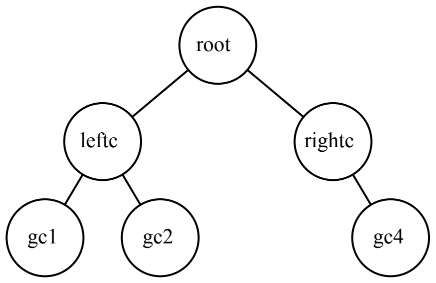
Sample Input	Output for the Sample Input
2-3-2-6-1 1-3-1-1-1 4-3-4-6-4 2-3-5-5-6 6-3-5-3-5 1-4-4-6-1 2-2-3-2-2 4-6-6-6-5 5-4-5-5-5 3-3-1-1-3 1-2-3-2-3 4-3-3-3-3 0-0-0-0-0	Case 1: 42 Case 2: 60

Problem 4: Describing Trees

Trees, particularly binary trees, are a common data structure, and describing them is a common task. One notation for describing a binary tree uses a triple (level, position, label) for each node. Level gives the depth of a node in the tree, with the root node at level 1, its children at level 2, and so forth. Position gives the position of the node among the nodes at the same depth, with the leftmost node at a given depth having position 1, the next node having position 2, and so forth. Label is the text to be used in labeling the node.

For example, the binary tree defined by the set of six input triples shown on the left would normally be illustrated as shown on the right.

```
(1, 1, root) (3, 1, gc1) (2, 2, rightc)
(2, 1, leftc) (3, 2, gc2) (3, 4, gc4)
```



Another representation of a binary tree is procedural. A node is specified using a line beginning with the word "node", followed by the node's position at the current depth and the label for the node. The initial depth is 1, and lines containing only the word "level" increase the tree depth by 1. Using this representation, the tree shown above would be specified like this:

```
node 1 root
level
node 1 leftc
node 2 rightc
level
node 1 gc1
node 2 gc2
node 4 gc4
```

In this problem you are to read a description of a tree using the "triple" notation or the procedural notation. If the tree is not viable – that is, there is a non-root node without a parent, or there are too many nodes on a level, display an appropriate message. If the tree is viable, then display the tree using the other notation (that is, the one not used for the input description). There will be no empty trees in the input, and no two nodes at the same level will have the same position.

Input

There will be multiple test cases to consider. Each case consists of a sequence of lines giving the description of a tree terminated by a line containing only whitespace. All tree descriptions will be syntactically correct, but all trees will not necessarily be viable. The last input case is followed by a line containing only whitespace. Node labels will consist only of alphanumeric characters, and will be no longer than 8 characters. In the triple notation for tree nodes, whitespace (blanks, tabs, and end of line characters) may appear anywhere except in the middle of the decimal integers giving the level or position of a node, or within a node's label. In the procedural notation, blanks or tabs may appear before or after any of the elements on an input line. No tree will have more than 32 levels.

Output

For each input case, first display the case number (1, 2, …). If the tree is viable, then display its representation using the notation not used to describe it in the input. Otherwise display a message indicating the tree is not viable. To display a tree using the "triple" notation, display one triple per line, ordering the triples first by level, and within a level by position. In the procedural notation, display node lines ordered by position. Use the format shown in the sample output.

Sample Input

```
   (1, 1, root)   (3, 1, gc1)   (2, 2, rightc)
      (2, 1, leftc)   (3, 2, gc2)   (3, 4, gc4)
blank line
               (2, 2, rchild)
   (1, 1, root)
blank line
   (2, 1, root)
blank line
 (1, 1, root)(2, 1, c1)  ( 2,2,c2) (2, 3, c3)
blank line
  node 1 root
level
node 2 rchild
node 1 lchild
blank line
level
node 1 root
blank line
blank line
```

Output for the Sample Input

```
Case 1:
   node 1 root
   level
   node 1 leftc
   node 2 rightc
   level
   node 1 gc1
   node 2 gc2
   node 4 gc4

Case 2:
   node 1 root
   level
   node 2 rchild

Case 3:
    Not viable: a non-root node has no parent.

Case 4:
    Not viable: too many nodes on a level.

Case 5:
   (1, 1, root)
   (2, 1, lchild)
   (2, 2, rchild)

Case 6:
    Not viable: a non-root node has no parent.
```

Problem 5: If I'd known it was a race, I'd have gone faster!

Folks sometimes get upset with Bill because he always obeys the speed limits when driving. He believes many drivers and even some law-enforcement officers are of the opinion that driving a few miles or kilometers per hour over the posted speed limit is okay.

To convince others that his behavior is reasonable, Bill wants to show others how small a benefit is obtained from this "gray-zone" speeding. The benefit most would attribute to speeding is getting to the destination more quickly. In this problem we're going to give Bill some information to support his argument.

Assume a driver travels on a straight road with a single traffic signal that cycles between green, yellow, and red (as usual). Given the position of the traffic signal on the road, the speed limit, and the time the signal is green, yellow, and red, we want to know the average decrease in time required to drive this segment of road if the driver speeds by a given amount. To make things simple, we'll assume the driver only stops for the red signal, and proceeds past the light if it is green or yellow. And the vehicle being driven is phenomenal! It can stop instantaneously, and can also instantaneously go from a stop to any arbitrary speed!

For example, consider a one kilometer road that effectively has no traffic signal (that is, assume the length of time the signal stays red is 0). If the speed limit is 60 kilometers per hour, then driving this road at the speed limit will require 1/60th of an hour, or 60 seconds. If the driver speeds through at 65 kilometers per hour, then it will take 1/65th of an hour, or about 55.4 seconds. So the driver saves an entire 4.6 seconds by speeding.

Input

There will be multiple input cases to consider. The input for each case is a line containing 7 integers giving the length of the road, the distance from the beginning of the road to the traffic signal, the length of time the signal is green, yellow, and red (in that order, each given in seconds), the speed limit, and the amount over the speed limit the driver drives. Distances are in kilometers, and speeds are in kilometers per hour. The line following the input for last case will contain 7 zeroes.

Output

For each case, display the case number and average number of seconds the driver will save by speeding, with one fractional digit. Answers that are within 0.1 second of the correct answer will be accepted.

Sample Input	Output for the Sample Input

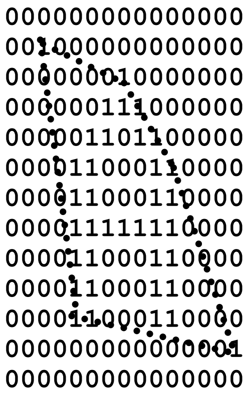
`1 0 1 0 0 60 5`
`20 5 27 3 30 60 5`
`0 0 0 0 0 0 0`
	`Case 1: 4.6`
	`Case 2: 92.3`

Problem 6: The Speckled Letter

With inexpensive scanners and computing systems, optical character recognition, or OCR, is now frequently used and very reliable. Most OCR systems perform some recognition tasks by matching the outline of a scanned character against the outline of known samples. Unfortunately, spots on a scanned image have a significant negative impact on this matching process. In this problem you'll consider how to "despeckle" the image of a character to remove these extraneous spots.

The image of a scanned character will be provided as a rectangular bitmap indicating the light and dark *pixels*, or picture elements, comprising the image. The dark pixels represent regions containing ink (or extraneous marks), and the light pixels represent the unmarked background on which the character was placed. For this problem, 0 and 1 represent light and dark regions, respectively. For example, on the left below is a bitmap with 13 rows and 15 columns representing the capital letter A with two extraneous pixels (speckles, or bad spots) that we wish to remove. If we number the rows 0 to 12 from the top to the bottom, and the columns 0 to 14 from the left to the right, then the extraneous pixels are at row 1, column 2 and row 11, column 14.

```
000000000000000
001000000000000
000000010000000
000000111000000
000001101100000
000011000110000
000011000110000
000011111110000
000011000110000
000011000110000
000011000110000
000000000000001
000000000000000
```



```
000000000000000
001000000000000
000000010000000
000000111000000
000001101100000
000011000110000
000011000110000
000011111110000
000011000110000
000011000110000
000011000110000
000000000000001
000000000000000
```

To identify the unwanted pixels, first determine the area of the smallest convex polygon that encloses all the dark pixels, treating each dark pixel as a point. This polygon can be visualized as that resulting from stretching a rubber band around the set of dark pixels in the bitmap. This is illustrated by the dotted lines superimposed on the bitmap shown on the right above.

Next determine if removing any of the dark pixels will reduce the area of the convex polygon by at least a specified percentage. If it does, assume that pixel was an unwanted "speckle", remove it from the bitmap (that is, set it to 0), and repeat the steps until there are no pixels that can be removed to reduce the area by at least the specified percentage. If there are multiple pixels that could be removed to suitably reduce the area, select the pixel that reduces the area by the largest percentage. Resolve any remaining tie by first removing the pixel with the smallest row number, and then the pixel with the smallest column number, assuming pixels are numbered as described above.

Input

There will be multiple input cases to consider. The input for each case begins with a line containing two integers *NR* and *NC* (1 ≤ NR, NC ≤ 100) giving the number of rows and columns in the bitmap, and a real number between 0 and 100 giving the percentage by which a pixel's removal must reduce the area of the enclosing convex polygon for it to be removed. This line is then followed by the bitmap, given as *NR* lines each containing *NC* characters, each 0 or 1, followed by the end of line character. The input for the last case is followed by a line containing three zeroes.

Output

For each case, display the case number (1, 2, …) and the list of pixels to be removed in the order they should be removed. If no pixels are to be removed, explicitly state that as shown in the sample output. Your output should be formatted as shown in the sample below.

Sample Input

```
13 15 5.0
000000000000000
001000000000000
000000010000000
000000111000000
000001101100000
000011000110000
000011000110000
000011111110000
000011000110000
000011000110000
000011000110000
000000000000001
000000000000000
13 15 5.0
000000000000000
000111110000000
000110011000000
000110001100000
000110100110000
000110001100000
000111111000000
000110001100000
000110000110000
000110100011000
000110000110000
000111111100000
000000000000000
0 0 0
```

Output for the Sample Input

```
Case 1:
   Delete pixel at (1,2)
   Delete pixel at (11,14)

Case 2:
   No pixels deleted.
```

Problem 7: Cell Phone Texting

Everybody now sends SMS ("text") messages from phone to phone. On many phones, the procedure for sending a message is to press the numeric key associated with the desired letter as many times as necessary in order to select that letter. For example, to create the letter "N", one would press the "6" twice – the first selects "M" and the second press changes the "M" into an "N". A space between words is created by pressing the "#" once.

This procedure is sufficient for a word like "BET" where one can first press "2" twice, then "3" twice, then "8". A complicating factor is when the adjacent letters use the same number, as in "CAB". It is necessary to press "2" three times for the "C", then wait a certain time, then press "2" once for the "A", wait a certain time, then press "2" twice for the "B". Let's refer to this waiting as a "pause" in the buttons, and the notation "222P-2P-22" stands for the word "CAB". There is a dash between the keys necessary for each letter, and there is a "P" to indicate the necessary pause.

Examples

HELLO	44-33-555P-555-666
THIS IS A MESSAGE	8-44P-444-7777-#-444-7777-#-2-#-6-33-7777P-7777-2-4-33

The keypad to be used for this problem is as follows:

1	2 ABC	3 DEF
4 GHI	5 JKL	6 MNO
7 PQRS	8 TUV	9 WXYZ
*	0	# space

The problem

Write a text message encoder/decoder. When presented with a plain text message, the program should print the text message equivalent. When presented with a text message, the program should print the plain text version. The program can determine whether to convert to or from text messages based on the initial character of the message; a number or "#" should convert from text format to plain format, and a letter or space should convert from plain text to text format. In the latter case, the input is guaranteed to be upper case letters and will contain no digits.

Input

There will be multiple lines in the input. Each line represents one message, and each message is to be read, converted as necessary, and displayed. The last line in the input will start with a "*" and indicates that the program should terminate.

You may assume all of the input is valid. There will be no invalid characters or tab characters in the input, and there will be no trailing spaces at the end of a line.

Plain text lines will contain only uppercase alphabetic characters, space characters, and the end of line.

Output

For each input line, create the appropriate output line.

Sample Input

```
THIS IS A MESSAGE
4-777-33P-33-8-444-66-4-7777-#-2P-222-6-#-222-666-3P-33-777P-7777
8-33-99-8
TEXT
*
```

Output for the Sample Input

```
8-44P-444-7777-#-444-7777-#-2-#-6-33-7777P-7777-2-4-33
GREETINGS ACM CODERS
TEXT
8-33-99-8
```

Problem 8: Edit Distance

The *edit distance* between two words (two sequences of alphanumeric characters) is defined as the minimum number of characters that need to be replaced, added, and/or deleted from the first word to transform it into the second word[1].

Your task in this problem is to find the edit distance between a given pair of words.

Examples

Consider the words "CAT" and "BAT". The two words only differ in their first character. We only need to replace the 'C' in "CAT" with a 'B' to arrive at "BAT". Therefore the edit distance is 1.

The words "FLY" and "FLYING" are identical in their first three characters, but the second word has three additional characters. Adding "ING" to the first word produces the second word. The edit distance in this case is 3.

Consider "GRAVE" and "GROOVY". We can perform the following substitutions in the first word: (1) 'A' -> 'O', (2) 'E' -> 'Y', then (3) insert the character 'O' in position 4 (after the first 'O'). Thus, the edit distance in this case is 3.

Input

The input will start with an integer *N* (*N* > 0) on a line by itself. This is followed by *N* lines, each of which contains a pair of words. Each word will be no longer than 255 characters, and will contain only non-space alphanumeric characters. The case of alphabetic characters is to be ignored.

Output

For each pair of words, output the pair's sequence number (starting from 1 for the first pair) followed by " Edit Distance = " and the value of the edit distance between the two words.

Sample Input	Output for the Sample Input

`5`
`Cat Bat`
`Fly Flying`
`Grave Groovy`
`Kitten Sitting`
`Alpha Omega`
	`1. Edit Distance = 1`
	`2. Edit Distance = 3`
	`3. Edit Distance = 3`
	`4. Edit Distance = 3`
	`5. Edit Distance = 4`

[1] Levenshtein Distance, http://en.wikipedia.org/wiki/Levenshtein_distance

Problem 9: The Encoding

Professor Moriarty has devised what he believes is a fiendishly clever code to communicate with his cohorts. At the first level, messages using this code are a sequence of numbers, each of which consists of between 2 and 9 digits (each digit in the range 1 through 9) in ascending and consecutive order (like 2345, or 789). To make things worse, at the second level of encoding he replaces each digit with a capital letter, and these letter sequences are then transmitted as the encoded message.

Luckily, some of the cohorts discarded a slip of paper containing a clue that enables you to decode the numeric sequences into the original messages. But you must still find some way of translating those letter sequences in the corresponding sequences of numbers. Sherlock has asked you to help.

As an example of what's needed, consider the message containing these 8 letter-sequences:

```
RE    EW    OIU    IUYTR    YTR    TRE    EW    POIU
```

After manipulating these for a while, Sherlock found they match the digits 1 through 9 as shown in the following table. But it takes too long to do messages by hand, so a program to do the job is required.

```
1 2 3 4 5 6 7 8 9
P O I U Y T R E W
```

Input

There will be multiple input cases to consider. The input for each case begins with a line containing an integer *N* which specifies the number of letter-sequences in a message. *N* will be no larger than 100. On the next *N* lines are those letter-sequences, each of which consists of between 2 and 9 uppercase alphabetic characters immediately followed by the end of line. The input for the last case is followed by a line containing a single zero.

Output

For each case, first display the case number. If it is impossible to uniquely determine the mapping from letters to digits, display the word "Ambiguous." Otherwise, display the mapping from letters to digits as shown in the sample below, displaying a blank line after the output for each case.

Sample Input

```
5
GL
OWIZXUGL
WIZX
WI
ZX
8
RE
EW
OIU
IUYTR
YTR
TRE
EW
POIU
0
```

Output for the Sample Input

```
Case 1:
   Ambiguous

Case 2:
   1 2 3 4 5 6 7 8 9
   P O I U Y T R E W
```