Scavenger Hunt

Joe likes to create scavenger hunts in his backyard to keep his children busy during the summer. He has created an 8x8 grid of land and buried something in each of the 64 cells. In some of the cells, he's buried prizes for the kids, such as a bag of candy, or the day's WiFi password.

In the rest of the cells he's buried a scrap of paper with a 2-digit number written on it. This number is a clue directing the kids to the next section of land to search. The tens digit indicates the row number and the ones digit indicates the column number. For example, 37 indicates row 3 and column 7. The rows and columns are both numbered 1–8.

His kids always begin the search in the upper-left hand corner of the grid, which is row 1, column 1. They then follow the clues until they either find a prize or are led back to a section of land they have already searched.

Your goal is to help Joe plan these hunts by writing a program which reads in what is buried in each section of land, and then simulates the search his kids will perform. At the end, you will know if the search leads to a prize or not.

Program input: The first line of input contains an integer N giving the number of test cases in the input. Following that are N test cases. There are no blank lines separating test cases. Each test case consists of 8 lines, giving the 8 rows of Joe's yard.

Each row contains 8 sets of 2 characters, separated by spaces, which indicate what is buried in each of the 8 columns in the row. These two characters are either two digits, which indicate a clue to another location, or "??" which indicates a prize is buried in this cell. You can assume that no clue will refer to a non-existent cell (such as 40 or 93).

Program output: For each test case, you should output the sequence of cells that the kids will search, represented as two-digit numbers, each on their own line. The first line of output should always contain "11" as that's where the search begins. The sequence ends with the first cell that either contains a prize or has already been visited. The last line of output for each test case should either be "Found a prize!" or "No prize found."

Example input:

```
2
87 55 86 88 76 48 61 43
84 73 17 52 73 27 66 43
33 12 84 21 81 31 72 11
57 32 21 14 76 33 83 24
73 37 58 55 75 37 14 68
76 24 54 38 56 21 33 62
48 76 45 83 75 ?? 21 72
27 46 15 22 83 24 51 81
17 15 73 14 24 13 86 33
81 54 ?? 82 37 56 21 85
11 75 32 86 83 53 16 18
82 75 37 83 44 45 ?? 23
78 11 13 18 68 82 32 53
37 78 86 16 86 45 43 22
21 84 27 82 ?? 72 23 31
85 86 86 22 47 46 12 87
```

Example output (corresponding to the input shown above):

```
11
87
51
73
45
76
Found a prize!
11
17
86
46
45
44
83
86
No prize found.
```

Jumbled Up

You work for the Bitburg Post-Dispatch newspaper as a puzzle developer. Your readers have demanded to have a “Jumble” type puzzle in the paper, wherein readers must un-scramble a series of scrambled words. For instance if the scrambled word is “TWHISC”, the savvy reader will realize the only word which can be made from these letters is “SWITCH”.

Not every word can be used in creating a Jumble puzzle. For instance the scramble “GANEL” cannot be used since there are three possible solutions: “ANGEL”, “ANGLE”, and “GLEAN”. A valid Jumble puzzle only has one possible solution, so the answer cannot have any anagrams which are also valid words.

In order to facilitate making Jumble puzzles, you need to write a program which will read in a set of words and find the ones which are “jumble-able” – that is the words for which there is no other word in the set of words containing the same set of letters.

Program input: Input consists of a number of test cases. Each test case begins with a line containing a number *N* giving the number of words in this test case. Following that is a line containing those *N* words, separated with spaces. A value of 0 for *N* indicates that there are no more test cases.

Program output: Output consists of one line containing the text "Results:", followed by a list of the words from the test case that are jumble-able, printed in alphabetical order.

Example input:

```
7
TASTE GLEAN ALERT ANGLE MOUNT ANGEL LATER
6
ZOO ALIGNED DEALING LEADING LEANING DEAL
0
```

Example output (corresponding to the input shown above):

```
Results:
MOUNT
TASTE
Results:
DEAL
LEANING
ZOO
```

Werewolf Transformations

You, for better or for worse, are a werewolf. But unlike a werewolf who is a mindless beast, powerless to resist the call of the full moon, you can transform back and forth from wolf form at will. Naturally you use this power to become a super hero: Wolf Person™.

In your wolf form you are are able to run much faster than you can in your regular, human form. However you struggle with other tasks as a wolf, such as opening doors (with no opposable thumbs and all). In particular, you can run 10 meters per second as a wolf, while only 4 meters per second as a human. It takes 8 seconds to open a door in wolf form and only 1 second as a human. It also takes 5 seconds to transform from one form to another.

As Wolf Person™, you often find yourself chasing suspects down long hallways with doors along it. You want to know the fastest way for you to run along such a hallway. Should you do it as human to open the doors faster, as a wolf to run faster, or will you need to transform back and forth to maximize your overall speed?

You should write a program that, given the distance you need to travel, and the locations of all of the doors in that distance, calculates the minimum time needed to cover the distance, along with how many transformations will be made.

You always begin the chase in your human form. You can be in either human or wolf form when you reach the end of the hallway.

Program input: The first line of input is an integer, N, giving the number of test cases. Following that will be N test cases, each of which is comprised of three lines. The first line of each test case is the distance, in meters, that you need to travel. The second line of each test case will contain the number of doors. There will be between 1 and 100 doors. The third line contains the locations of the doors along that path, as floating-point numbers separated by spaces.

Program output: Your output for this program should consist of two numbers for each test case, separated by a space. The first of these is a floating-point number, with two decimal points of precision given, indicating the minimum number of seconds needed to make it through the hallway. The second is an integer giving the number of transformations made along the way.

Example input:

```
3
100.0
4
10.0 80.0 85.0 90.0
25.0
7
5.0 8.5 10.0 15.5 21.0 22.75 24.0
```

```
750.0
2
300.0 500.0
```

Example output (corresponding to the input shown above):

```
28.50 2
13.25 0
96.00 1
```

500

There is a popular children's game called "500" which is played with one ball and at least three players. One of the players is designated as the thrower. This player throws the ball up in the air towards the other players. While throwing the ball, the thrower also shouts out a point value. Whoever catches the ball then receives that many points. If the ball hits the ground without being caught, then nobody receives those points. This continues until one player reaches 500 points, which makes them the winner.

Instead of calling out a numeric value for a throw, the thrower can opt to shout "Jackpot!" which means that anyone who catches that throw immediately wins. They can also instead shout "Bankrupt!" which means that whoever catches that throw loses all of the points they've accrued, going back to 0.

Sometimes the players lose track of how many points they have and so your task is to write a program which will decide the winner of games of 500, given the sequence of throws that take place. Keep in mind that it's possible that a game will continue a bit after someone has won. Any throws that take place after a player has won should not have any effect on the outcome.

Program input: The input consists of multiple games. The first line of input for each game is an integer N giving the number of throws in that game. Following that will be N lines, with each corresponding to one throw of the ball. Each of these lines consists of two parts, separated by a space.

The first is the value called out for the throw. This will either be a positive integer less than 500, "BANKRUPT" or "JACKPOT". The second part of each throw line contains either the name of the player who caught the throw, or "GROUND" if the ball hits the ground before anyone could catch it.

If N is 0, that indicates that there are no more games to process.

Program output: The output should consist of one one line for each game of 500 in the input. Each of these output lines should contain either "The winner is " followed by the name of the winner, or the text "There is no winner." if none of the players reached five hundred points.

Example input:

```
10
150 Mark
200 GROUND
75 Stephanie
100 GROUND
200 Stephanie
BANKRUPT Mark
325 Luis
400 Mark
JACKPOT GROUND
250 Stephanie
```

```
3
150 Mark
JACKPOT GROUND
50 Stephanie
0
```

Example output (corresponding to the input shown above):

```
The winner is Stephanie.
There is no winner.
```

Lawn Mower Control

Susan has just purchased a new XQ7-1000 automatic lawn mower to mow her yard. These mowers work with the help of a control program which tells the mower how to move through the yard. Once the control program is activated, the mower follows the instructions therein, leaving a trail of beautifully cut grass and its owner free to enjoy a leisure-time activity of their choice. At the end of the program, the mower halts and turns itself off.

These mowers operate as stack-based machines where each instruction in the control program causes the mower to take some action. The stack of the mower is initially empty and can hold up to 100 integer values. The XQ7-1000 model supports the following instructions in its control programs:

Instruction	**Meaning**
push [value]	Pushes the given integer value onto the stack.
add	Pops the top two values off the stack and adds them, putting the result on the stack.
move	Pops the top value off the stack and moves the mower by that number of meters. If the value is positive, the mower should move forward; if negative, it should move backward.
turn	Pops the top value off the stack and turns the mower by that angle. The value will always be a multiple of 90. Positive values turn the mower to its left and negative values turn it to its right.
branch [line]	If the top value on the stack is non-zero, control transfers to the given line of the program. If the top value is zero, the instruction does nothing. This instruction does not modify the stack. Lines are numbered starting from 0.
halt	Stops the mower and ends the control program.

When the mower begins operation, it starts by executing the first line in the control program, and continues on executing them one by one until it halts. Along the way, the move and turn instructions will cause the mower to move throughout its environment.

Susan has written several of these control programs for the new mower, and wants to know where each one will leave the mower when they eventually end. Your job is to write a program that will read in a mower control program and report the final position of the mower once it has finished, relative to where it starts.

When the mower begins execution, it is at the (0, 0) origin point of a coordinate system, facing in the positive direction along the Y-axis. You may assume that the control program consists of fewer than 100 lines, and will always halt eventually. You may also assume that each branch instruction will specify a valid line number.

Program input: The input consists of multiple control programs. The first line of input for each program will consist of an integer *N*, giving the number of lines in the program. Following will be *N* lines, each giving one line of the control program in the format given above. If *N* is 0, that indicates there are no more control programs to test.

Program output: The output should consist of one line for each control program being tested. Each of these output lines should report the final position of the mower relative to where it started in (X, Y) coordinates.

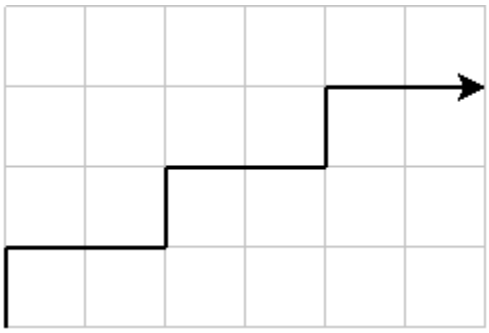
Example input:

```
13
push 3
push 1
move
push 270
turn
push 2
move
push 90
turn
push -1
add
branch 1
halt
0
```

Example output (corresponding to the input shown above):

```
Mower halts at (6, 3).
```

Visualization of Example input:

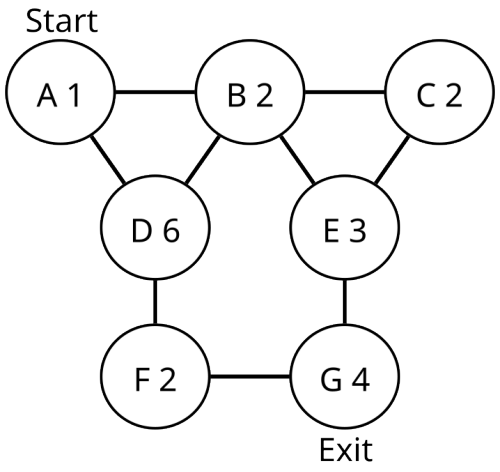


Cave Escape!

Archaeologist Montana Smith is exploring a cave system, in search of a lost artifact. The cave system consists of a number of open rooms connected together by passageways. Unfortunately at some point Montana steps on a hidden pressure plate trap, which triggers a collapse in the cave system! Now he must escape before he is crushed in the ensuing cave in.

The rooms of the cave system collapse on themselves at different times. In his escape, Montana needs to reach the room containing the exit without going through a room which has already collapsed, or being in a room while it collapses. You need to write a program which will read in the structure of the cave system, including when each room collapses, and determine if and how he can reach the cave's exit.

Below is an example cave structure. Montana begins in room A and must get to the exit, room G. The numbers refer to the step after which the room collapses. So room A will collapse after Montana's first step, and room B after his second. Montana always takes 1 step to travel from one room to another.



Montana can reach rooms B and D immediately. If he goes to room D, he will not be able to reach the exit because room F will collapse after his second move (when he would reach room F), so he must start by going to room B. Likewise Montana cannot continue to room C, but must escape via rooms E, then finally G. This case appears as the sample input below.

It is possible that there will be no escape from the cave system. For example if room E were to collapse after step 2 instead of 3, then Montana would not be able to reach it in time and there would be no possible escape for him.

Program input: The first line of input is a number N giving the number of cave systems in the input set. Each of the N cave systems begins with an integer R giving the number of rooms in the cave. Following are R lines, one for each room. Each of these lines contains an upper-case character which gives the name of the room, and an integer giving the step after which the room collapses. There will be no more

than 26 rooms, and no room will collapse before time step 1. The first room listed is the one in which Montana begins, and the last room is the one he must reach in order to escape.

Next is a line containing an integer *P* giving the number of passageways in the cave system. Following that are *P* lines which give the passageways. Each of these lines consists of two characters, indicating which two rooms the passageway connects. The passageways are bi-directional and only listed for one direction -– so BC indicates that there is a passageway that can be used to travel from room B to C or from C to B.

Program output: You should output one line for each of the *N* cave systems in the input. If there is a way for Montana to escape the cave system, your program should print "Montana can escape: " followed by the sequence of moves by which Montana can escape the cave. The sequence should be given as a string of upper-case characters which indicate which rooms Montana travelled through (including the starting room and the exit). If there are multiple solutions, you may print any of them. Your program should output "There is no escape!" if there is no escape from a cave system.

Example input:

```
1
7
A 1
B 2
C 2
D 6
E 3
F 2
G 4
9
AB
AD
BC
BD
BE
CE
DF
EG
FG
```

Example output (corresponding to the input shown above):

```
Montana can escape: ABEG
```

Cooking with Fractions

Julia has started a recipe blog to share her amazing baking recipes with her followers. She has the recipes all ready and has written novella-length introductions for each recipe for her readers to scroll past. The only thing she needs help with is a feature that lets her readers multiply the recipe ingredients. This way if they want to double or triple the yield of the recipe, they can do so automatically.

You should help Julia by writing a program which will read in the amount to multiply the recipe by and a list of ingredients, and output the revised ingredient list with larger amounts. The amount to multiply will always be an integer and always be greater than one (so the amounts will only ever be increased, not decreased).

The amounts given in the ingredient list can appear as integers, fractions, or "mixed numbers" containing both a whole number and a fraction component. For example, the following are all valid ingredient amounts:

```
2 cups potatoes, peeled and chopped
3/4 teaspoon cloves
1 1/2 jars of marinara sauce
```

Your program should output the multiplied ingredient list using the same formats and in the simplest terms possible. For instance, if we were to double the ingredient list above, we should get the following:

```
4 cups potatoes, peeled and chopped
1 1/2 teaspoons cloves
3 jars of marinara sauce
```

Notice that the fraction `3/4` became a mixed number when doubled and the mixed number `1 1/2` became a whole number. Your program should never list a fraction which can be simplified, so for example `2/4` should be written as `1/2` instead. Likewise, you should not list something as a fraction which could be written as a whole number instead, so `4/2` should just be written as `2`. Finally, no fraction should have a numerator which exceeds its denominator, so `8/3` should be written as the mixed number `2 2/3` instead.

We may also need to add pluralization to an ingredient list. The rule for this is that all ingredients that are 1 unit or less should not be pluralized, while those of greater than 1 unit should be. Notice that `3/4 teaspoon` is not pluralized, while `1 1/2 teaspoons` is. To pluralize an ingredient, we can simply add an 's' to the first word after the amount. You can assume that the input will use correct pluralization.

Program input: The first line of input contains two integers, K and N, separated by a space. K is the amount we are multiplying each ingredient by, and will be an integer greater than 1. N is the number of ingredients in the recipe. Following this, there will be N lines of input, each giving one ingredient, in the format described above.

Program output: Output should consist of N lines, one for each of the ingredients in the input. Each should have its amount multiplied by K.

Example input:

```
3 12
1 cup butter, softened
3/4 cup white sugar
1 cup packed brown sugar
2 eggs
2 teaspoons vanilla extract
1 1/2 teaspoons baking soda
1/4 teaspoon lemon juice
1/2 teaspoon salt
1/3 teaspoon hot water
3 cups all-purpose flour
2 1/4 cups semisweet chocolate chips
1 1/3 cups chopped walnuts
```

Example output (corresponding to the input shown above):

```
3 cups butter, softened
2 1/4 cups white sugar
3 cups packed brown sugar
6 eggs
6 teaspoons vanilla extract
4 1/2 teaspoons baking soda
3/4 teaspoon lemon juice
1 1/2 teaspoons salt
1 teaspoon hot water
9 cups all-purpose flour
6 3/4 cups semisweet chocolate chips
4 cups chopped walnuts
```