2016 University of Virginia High School Programming Contest

Welcome to the 2016 University of Virginia High School Programming Contest. Before you start the contest, please be aware of the following notes:

Rules

1. There are ten (10) problems in this packet, using letters A-J. These problems are *loosely* sorted by difficulty. As a team's solution is judged correct, the team will be awarded a balloon. The balloon colors are as follows:

Problem	Problem Name	Balloon Color
A	Service with a Smile	light blue
B	Omelette du Fromage	light purple
C	Stacks of Pancakes	gold
D	Regular Customers	dark green
E	Break-faster Delivery	orange
F	An Efficient Kitchen	dark blue
G	Waffle Coverage	red
H	Clumsy	pink
I	Breakfast Stations	yellow
J	A "Complete" Breakfast	dark purple

2. Solutions for problems submitted for judging are called runs. Each run will be judged.

   The judges will respond to your submission with one of the following responses. In the event that more than one response is applicable, the judges may respond with any of the applicable responses.

Response	Explanation
Yes	Your submission has been judged correct.
No - Wrong Answer	Your submission generated output that is not correct or is incomplete.
No - Output Format Error	Your submission's output is not in the correct format or is misspelled.
No - Excessive Output	Your submission generated output in addition to or instead of what is required.
No - Compilation Error	Your submission failed to compile.
No - Run-Time Error	Your submission experienced a run-time error.
No - Time Limit Exceeded	Your submission did not terminate within one minute.

3. A team's score is based on the number of problems they solve and penalty minutes, which reflect the amount of time and number of incorrect submissions made before the problem is solved. For each problem solved correctly, penalty minutes are issued equal to the time at which the problem was solved plus 20 minutes for each incorrect submission. No penalty minutes are added for problems that are never solved. Teams are ranked first by the number of problems solved and then by the fewest penalty minutes.

4. This problem set contains sample input and output for each problem. However, the judges will test your submission against several other more complex datasets, which will not be revealed until after the contest. One challenge is designing other input sets for yourself so that you may fully test your program before submitting your run. Should you receive a "wrong answer" judgment, you should consider what other datasets you could design to further evaluate your program.

5. In the event that you think a problem statement is ambiguous or incorrect, you may request a clarification. Read the problem carefully before requesting a clarification. If the judges believe that the problem statement is sufficiently clear, you will receive the response, "The problem statement is sufficient; no clarification is necessary." If you receive this response, you should read the problem description more carefully. If you still think there is an ambiguity, you will have to be more specific or descriptive of the ambiguity you have found. If the problem statement is ambiguous in specifying the correct output for a particular input, please include that input data in the clarification request.

   You may not submit clarification requests asking for the correct output for inputs that you provide. Sample inputs may be useful in explaining the nature of a perceived ambiguity, e.g., "There is no statement about the desired order of outputs. Given the input: ..., would not both this: ... and this: ... be valid outputs?".

   If a clarification that is issued during the contest applies to all the teams, it will be broadcast to everybody.

6. Runs for each particular problem will be judged in the order they are received. However, it is possible that runs for different problems may be judged out of order. For example, you may submit a run for B followed by a run for C, but receive the response for C first.

   Do not request clarifications on when a response will be returned. If you have not received a response for a run within 30 minutes of submitting it, **you may have a runner ask the site judge to determine the cause of the delay. Under no circumstances should you ever submit a clarification request about a submission for which you have not received a judgment.**

   If, due to unforeseen circumstances, judging for one or more problems begins to lag more than 30 minutes behind submissions, a clarification announcement will be issued to all teams. This announcement will include a change to the 30 minute time period that teams are expected to wait before consulting the site judge.

7. The submission of abusive programs or clarification requests to the judges will be considered grounds for immediate disqualification. This includes submitting dozens of runs within a short time period (say, within a minute or two).

Your Programs

8. All solutions must read from standard input and write to standard output. In C this is `scanf()` / `printf()`, in C++ this is `cin` / `cout`, and in Java this is `System.in` / `System.out`. The judges will ignore all output sent to standard error (`cerr` in C++ or `System.err` in Java). You may wish to use standard error to output debugging information. From your workstation you may test your program with an input file by redirecting input from a file:

   ```
   program < file.in
   ```

9. All lines of program input and output should end with a newline character (`\n`, `endl`, or `println()`).

10. All input sets used by the judges will follow the input format specification found in the problem description. You do not need to test for input that violates the input format specified in the problem.

11. Unless otherwise specified, all lines of program output should be left justified, with no leading blank spaces prior to the first non-blank character on that line.

12. Unless otherwise specified, all numbers in your output should begin with a '-' if negative, followed immediately by 1 or more decimal digits. If it is a real number, then the decimal point should be followed by as many decimal digits as can be printed. This means that for floating point values, use standard printing techniques (cout and System.out.println). Unless otherwise noted, the judging will check your programs with 10^{-3} accuracy, so only consider the sample output up until that point.

    In simpler terms, neither scientific notation nor commas will be used for numbers, and you should ensure you do not round or use a set precision unless otherwise specified in the problem statement.

13. If a problem specifies that an input is a floating point number, the input will be presented according to the rules stipulated above for output of real numbers, except that decimal points and the following digits may be omitted for numbers with no fractional component. Scientific notation will not be used in input sets unless a problem statement explicitly specifies it.

Good luck, and HAVE FUN!!!

A. Service with a Smile

Description

After slogging through four hours of coding difficult algorithms, four champion coders have decided to get a late-night meal at one of their favorite diners, the House Serving Pancakes and Coffee (or HSPC, for short). They each ordered a large serving of breakfast-for-dinner, and now are faced with the bill. Unfortunately, all of them left their slide rules, abacuses, and trig tables at home, and the restaurant is experiencing a computer malfunction, so no one is sure how much the whole meal should cost. And on top of that, our four friends really want to give their server a generous tip because of their outstanding service. While each of them *could* write their own program to figure it out, they're after four hours already spent coding! Your job is to write their program for them.

Input Format

The first line of input will be an integer, T, equal to the number of test cases that will follow. Each test case will start with a line with three integers: M, N, and P. M represents the number of items on the menu; N represents the number of items ordered by the four diners; and P represents the percent of the tip (given as an integer) — the percent of the total cost of the meal that the diners would like to pay their server, in addition to their bill. Following that will be M lines, with line i (starting with 1) containing an integer C_i equal to the cost (in cents) of item i. (People order by number at the HSPC). Then there will be N lines consisting of a single integer j signifying that one order of item O_j was ordered. Each menu item may show up zero, one, or more times in the list of orders. If it shows up more than once, then each time it appears corresponds to one order.

Output Format

For each test case, one should output "Case", the case number (starting with 1), a colon, and a space. Then, on that same line, output the total cost of the order (including the tip) in *dollars*. This means that one should begin the cost with a dollar-sign ($) and always have at least one value (possibly the value 0) before the decimal point followed by exactly two digits after the decimal point. If there are rounding issues from the tip calculation, then one should round up to the nearest cent (because champion coders are generous).

Sample Input

```
4
10 12 20
100
399
1499
1699
799
500
349
1199
1399
899
```

```
1
1
1
1
3
4
8
9
10
10
10
10
1 1 10
70
1
1 1 10
199
1
1 1 10
200
1
```

Sample Output

```
Case 1: $117.51
Case 2: $0.77
Case 3: $2.19
Case 4: $2.20
```

B. Omelette du Fromage

Description

You work as an assistant in a research lab, and there are some important calculations that the head researcher needs you to carry out. Unfortunately, the head researcher recently had a dream that has caused him to only be able to speak by repeatedly saying the phrase "omelette du fromage"! In order to make sure he understands the answers to the problems you need to phrase all the answers using only the phrase "omelette du fromage". Fortunately, we can express any positive integer by repeating "omelette du fromage" that many times! We call this "omelette du fromage notation."

Input Format

The input will begin with a single positive integer representing the number of test cases to follow. Each test case will consist of positive integers and the mathematical operators + and * (representing addition and multiplication). All input will be space separated. All input will be well-formed mathematical expressions.

Output Format

For each test case, you need to print out the test case number (in "omelette du fromage" notation), a colon, a space, and the answer to the math problem (in "omelette du fromage" notation). Each output value will be less than 2^{10}.

Sample Input

```
3
1 * 2 + 1
1 * 1
1 + 1
```

Sample Output

```
omelette du fromage: omelette du fromage omelette du fromage omelette du fromage
omelette du fromage omelette du fromage: omelette du fromage
omelette du fromage omelette du fromage omelette du fromage: omelette du fromage omelette du fromage
```

C. Stacks of Pancakes

Description

We've made a lot of pancakes and stacked them all on one plate. However, we've realized too late that all the pancakes were put on the wrong plate! Now we must move the entire stack from the current plate to the right one.

This is a bit difficult, though, since pancakes must always be in stacks of decreasing size–otherwise, the stack is unstable and pancakes will fall (tragic!). Because moving more than one pancake at a time makes it easy to drop pancakes, we must only move pancakes one at a time from the top of a stack to the top of another stack (or an empty plate). Moving pancakes one at a time means that we need at least one additional plate as a temporary storage place.

Can you help determine the number of movements necessary to transfer the pancakes from the original plate to the target plate?

Input Format

Each input file will start with a single integer $1 \leq T \leq 10000$ denoting the number of testcases. The following T lines each contain a single integer $1 \leq n \leq 60$, which is the number of pancakes on the original plate.

Output Format

For each case, print "Case ", followed by the case number, a colon, and then the number of pancake movements needed to move the pancakes from the original plate to the target plate.

Sample Input

```
2
1
3
```

Sample Output

```
Case 1: 1
Case 2: 7
```

D. Regular Customers

Description

There are many regular customers at the America's Cheap Meals breakfast house. A few of them even order the same thing every day! In order to improve efficiency, the chef has decided to make the meals for these customers as soon as they arrive; then the kitchen is slightly less affected by rushes and these customers are served faster. Given a historical list of customer names and orders, determine for certain customers whether or not you can predict their order. If the customer has at least 3 visits, and has ordered the same thing each time, then the customer is a regular who always orders the same thing - so you should print out their meal. If the customer has less than 3 visits, print "Not a regular", and if the customer has 3 or more visits but doesn't always order the same thing, print "Not a regular regular".

Input Format

Each input file will start with a single integer $1 \leq T \leq 10000$ denoting the number of testcases. Each test case will start with an integer $1 \leq N \leq 1000$. N lines will follow, starting with the customer's name (the name will contain no whitespace) and followed by their order. The next line will contain a single integer $1 \leq M \leq N$ denoting the number of customers. Each of the next M lines will contain the name of a customer — the output should be in the same order as these names.

Output Format

For each case, print "Case", a space, the case number, a colon, a new line, and then M lines containing each customer's name and their prediction.

Sample Input

```
2
8
Derek Hot Chocolate and Blueberry Bagel with plain cream cheese
Andrew Omelette with Green Peppers and Basil
Derek Hot Chocolate and Blueberry Bagel with plain cream cheese
Andrew Pancakes with Syrup
Derek Hot Chocolate and Blueberry Bagel with plain cream cheese
Martin Eggs and Bacon
Derek Hot Chocolate and Blueberry Bagel with plain cream cheese
Andrew Chicken
3
Derek
Andrew
Martin
3
Marina Waffles
Marina Waffles
Marina Waffles
1
Marina
```

Sample Output

```
Case 1:
Derek Hot Chocolate and Blueberry Bagel with plain cream cheese
Andrew Not a regular regular
Martin Not a regular
Case 2:
Marina Waffles
```

E. Break-faster Delivery

Description

In order to better provide service for lots of busy people, Elliot's Eatery has started a breakfast delivery program. However, there's a bit of a snag. Since Elliot's is located in a big city, there are many one-way streets, some of which are just one-way during the morning rush hour (when most of the breakfasts need to be delivered). Further, at that time of day some streets are practically unavailable due to high traffic volume! This is problematic, since it is not guaranteed that it is possible to actually deliver all of the breakfasts and get back to the eatery! In order to save the hassle, Elliot's Eatery needs a system capable of verifying whether or not it's actually possible to deliver to every address and get back — and they're asking you to build it! The map is handled as a set of intersections, along with pairs of intersections (a b), designating that it is possible to drive from a directly to b along a one-way road. Two-way roads are designated with two pairs (essentially as one one-way road in each direction).

Input Format

The first line of input will consist of a single integer T that is equal to the number of test cases that will follow. Each test case, will begin with one line consisting of three integers I ($1 \leq I \leq 20,000$), R ($1 \leq R \leq 10 \cdot I$), and L ($1 \leq L < I$) specifying the number of intersections, one-way roads, and locations that need delivery, respectively. Following that line will be R pairs of integers. Each pair is the indices of a source and destination intersection for a one-way road (intersection numbers ranging from 0 to $I - 1$ inclusive). The next L lines will consist of a series of customer locations (also identified by intersection number) that might or might not be accessible from the eatery. The eatery is always at position 0.

Output Format

For each test case, output exactly one line of output beginning with the word "Case", a space, the case number (starting at 1), a colon, and another space. Then, if every customer was accessible from the eatery, output "All are accessible" on that same line. If not, output the intersection number of the inaccessible customer with the smallest intersection number.

Sample Input

```
2
3 3 1
0 1 1 2 2 0
2
3 3 1
0 1 1 0 2 0
2
```

Sample Output

```
Case 1: All are accessible
Case 2: 2
```

F. An Efficient Kitchen

Description

The breakfast rush can be a very busy time. In order to improve their service during the rush, Casey's Café is trying a novel way of moving food around the kitchen quickly: instead of carrying plates of food around, the chefs carefully launch the food so it lands on the target plate safely! Although this plan has some critics, it has the potential to greatly reduce the waiting times in the restaurant. This relies on whether or not the chefs can quickly determine the angle and speed at which to launch the food.

Target plates are directly in front of the launcher, so they may be described in x and y coordinates (with no need for a 3rd dimension). Given target locations, your program must determine the initial speed and angle the food should be launched at so it reaches the target with the minimum possible y velocity.

The following facts from physics may be useful:

- Gravitational acceleration is $g = 9.81\frac{m}{s^2}$.
- Projectiles launched from the position (x_0, y_0) move according to the following two equations:

$$y = -\frac{1}{2}gt^2 + v_{y_0}t + y_0$$
$$x = v_{x_0}t + x_0$$

  where v_{x_0} and v_{y_0} are the initial x and y velocities, g is gravitational acceleration, and t is the time since launch.

- Projectile velocity acts according to the following two equations (using the same variables as before):

$$v_y = -gt + v_{y_0}$$
$$v_x = v_{x_0}$$

  That is, the y-component of velocity changes linearly, while the x-component is constant.

- The maximum height a projectile launched from $(0, 0)$ may attain is

$$y_{max} = \frac{v_{y_0}^2}{2g}$$

  At this point, the y-component of velocity is 0.

- The speed of an object with component velocities v_x and v_y is $\sqrt{v_x^2 + v_y^2}$ and its angle is given by $\theta = \arctan\left(\frac{v_y}{v_x}\right)$

The x and y distances to the target are measured in meters and gravity is assumed to pull towards the negative y direction.

Input Format

Each file begins with a single integer, T, which is the number of test cases to follow.

Each test case consists of a single line with three real numbers m, x, and y. The mass of the projectile is denoted by $0 < m \leq 10$, while the numbers $0 < x \leq 10^9$ and $-10^9 \leq y \leq 10^9$, $y \neq 0$, describe the distance from the launcher to the target plate in meters in each coordinate direction. (The launcher is assumed to be at $(0, 0)$).

Output Format

Output each case starting with "Case", a space, then test case number followed by a colon and a space. On the same line, print the initial speed, v, (in meters per second) and the angle, θ, (in degrees) of launch measured from the positive x axis ($-180^\circ < \theta < 180^\circ$) separated by a space. Both numbers should be printed to two decimal places.

Sample Input

```
3
1 1 1
7.34 77 57
6.15 7.8 -2.5
```

Sample Output

```
Case 1: 4.95 63.43
Case 2: 40.36 55.96
Case 3: 10.93 0.00
```

G. Waffle Coverage

Description

In an attempt to get (rectangular) waffles distributed more quickly, a device has been constructed that automatically pours maple syrup on waffles. It does this by making sequential passes over columns and/or rows of waffles. Note that waffles aren't perfect grids, though! Due to slight imperfections in the waffle-crafting process, often a waffle will have one or more cells that aren't completely separated. In order to get the best syrup coverage on each waffle, we want to take advantage of this fact to get as many cells covered in the limited number of passes we can do before the waffle gets too cold to serve.

Input Format

Input starts with a single positive integer indicating the number of test cases.

Each test case begins with three numbers, denoting the width $1 \leq W \leq 50$ and height $1 \leq H \leq 50$ of the waffle, and the number of passes $1 \leq P \leq 3$ made by the syrup machine. The next $2H + 1$ lines describe the waffle as an ASCII grid.

The odd lines of this grid description describe the horizontal separating structures of the waffle, while the even lines describe the vertical separating structures. If a '-' is replaced by a space, or a '|' is replaced by a '.', that means that the separating structure in that position is no longer there (a baking defect).

The passes made by the machine go directly through the center of a row or column of cells, and hit every cell on the path, filling it and any cells to which it is connected by a series of baking defects. The outside edge of any waffle will never have a baking defect.

Output Format

As output, print the word "Case", a space, the case number, a colon, a space, and then the maximum number of cells that can be filled the given number of passes.

Sample Input

```
4
4 4 3
+-+-+-+-+
| | | | |
+-+-+-+-+
| | | | |
+-+-+-+-+
| | | | |
+-+-+-+-+
| | | | |
+-+-+-+-+
4 4 2
+-+-+-+-+
| | | | |
+-+-+-+-+
| | . | |
```

```
+-+ +-+-+
| . | | |
+-+-+-+-+
| | | | |
+-+-+-+-+
4 4 3
+-+-+-+-+
| | | | |
+-+-+-+-+
| | . | |
+-+ +-+-+
| . | | |
+-+-+-+-+
| | | | |
+-+-+-+-+
4 3 2
+-+-+-+-+
| | | | |
+-+-+-+-+
| | . | |
+-+ +-+-+
| . | | |
+-+-+-+-+
```

Sample Output

```
Case 1: 12
Case 2: 11
Case 3: 14
Case 4: 10
```

H. Clumsy

Description

We have had a tough day making crepes (a thin, French pancake) and have placed all of them in one giant stack. However, apparently we are a little clumsy and don't have the best stack discipline, because as we are moving the crepes back from the crepe-making station to the table, someone accidentally trips and all of our crepes come falling onto the ground! We resign ourselves to having to pick all of them back up, but we've realized that we can make the job go faster if we can pick them in groups. In fact, we realize we have the ability to pick up any group of crepes so long as each crepe in the group touches at least one other crepe in that same group.

Crepes, because they are so thin, and because we are very good at making them round, can be modelled as perfect circles. However, we are rather inconsistent, so each crepe might be a different size. Two crepes then touch if and only if their representative circles intersect at at least one point.

Input Format

The first line of input will be the number T consisting of the number of test cases. For each test case, the first line will have the number of crepes M ($1 \leq M \leq 15,000$) that we have dropped in that test case. Following this will be M lines with three integers X_i, Y_i, and R_i, specifying the x- and y-coordinates as well as the radius of the i_th crepe, where $-150,000 \leq X_i \leq 150,000$, $-150,000 \leq Y_i \leq 150,000$, and $1 \leq R_i \leq 3,000$.

Output Format

For each input, there should be one line of output. This should consist of the word "Case", a space, the case number, a colon, a space, and then the number of different groups of crepes that we will have to pick up.

Sample Input

```
1
5
0 0 3
1 1 3
2 1 2
6 1 1
4 4 1
```

Sample Output

```
Case 1: 3
```

I. Breakfast Stations

Description

You're visiting a breakfast buffet with your classmates. The buffet is divided up into several stations, each of which serves a particular food item. Each of you have preferences for certain breakfast items, but no one wants to eat the same food as anyone else. You want to find whether you and your friends may eat with this restriction.

Input Format

The input will begin with a line containing a single integer T, which indicates the number of test cases.

Each test case will begin with a line indicating the number of food types, $1 \leq f \leq 500$. The following f lines will each consist of the name of one food.

Next, there will be a line containing a single integer $1 \leq s \leq 500$, which indicates the number of students. The following s lines each start with a name, followed by an integer number of foods they like to eat, and then a list of these foods.

All names (for foods and for people) consist of alphabetic characters with no spaces (e.g. "EggsBacon" is valid, "Eggs and Bacon!" is not).

Output Format

For each case, output "Case", a space, the case number, a colon, then the solution to the case as follows: If there is no matching such that every student eats a distinct food, print "NOT_POSSIBLE". Otherwise, print "POSSIBLE".

Sample Input

```
2
2
Pancakes
Omelette
3
Andrew 1 Pancakes
Martin 1 Omelette
Marina 1 Omelette
4
Pancakes
Omelette
Chocolate
Juice
4
Andrew 2 Pancakes Omelette
Martin 1 Omelette
Marina 1 Chocolate
Karen  2 Juice Pancakes
```

Sample Output

```
Case 1: NOT_POSSIBLE
Case 2: POSSIBLE
```

J. A "Complete" Breakfast

Description

The problem of waiting tables in a breakfast house is incredibly hard. A significant amount of this complexity comes from the fact that customers are getting increasingly more complex in their orders, taking the troubled waitstaff more and more time to write them down accurately and completely. In an attempt to remedy this, the House Serving Pancakes and Coffee (HSPC) has developed a new shorthand for waitstaff to encode customers' orders. This should allow them to take even the most complex orders using only a tiny fraction of the space! Each order is represented by a hexadecimal string, which can then be processed by the kitchen to get the exact order that the customer requested.

The scheme works as follows: 16 unsigned 8-bit variables are kept in the working set, of which the 0th indicates the position in the recipe currently being worked on. All start at 0. The recipe is read into a 256-byte structure, the rest of which is initialized to 0 The scheme also involves a stack of unsigned 8-bit variables; this acts as a call stack. Instructions are given a code in a single byte, and depending on that byte potentially a few subsequent bytes are used as part of the instruction (as arguments). Instructions are aligned on byte boundaries. In the following notation, each character represents a 4-bit nybble and the same character twice meaning a full-byte value. The instructions are:

```
00 xy: var[x] = var[y]
01 0x yy: var[x] = y
02 xy: recipe[var[x]] = var[y]
03 xy: var[x] = recipe[var[y]]
04 xy: var[x] += var[y]
05 xy: var[x] -= var[y]
06 xy: var[x] &= var[y] (& is the bitwise "and" operation)
07 xy: var[x] |= var[y] (| is the bitwise "or" operation)
08 xx: push(var[0]+2); var[0] = xx
09: var[0] = pop()
0A xy zz: if(var[x] == var[y]) var[0] = zz
0B xy zz: if(var[x] > var[y]) var[0] = zz
0C zz: var[0] = zz
0D 0x: var[x] = getchar() (get one character from the input and put it in var[x],
       or put 0 in var[x] if the input is exhausted)
0E 0x: print((char)var[x]) (cast var[x] to an ascii character, and print it out
0F: terminate program (attempting to execute an instruction at position 255
       also terminates the program cleanly)
```

Input Format

Input starts with a positive integer T, denoting the number of test cases. Each test case is two lines. The first line of each test case is a hexadecimal string representing the order. The second line of each test case is the input to be consumed during execution of the order (although it doesn't need to be used). The programs are valid and execute successfully under the presented rules. An explanation of the given test case is below:

```
010111 // var[1] = 17, the index at which the string starts
```

```
010701 // var[7] = 1
// start of loop
0321 // var[2] = recipe[var[1]]
0A231D // if var[2] == var[3] (which is 0), jump to terminator at position 29
0E02 // print var[2]
0417 // var[1] += 1
0C06 // jump to start of loop at position 6
48656c6c6f20576f726c6400 // Hello World\0
0F // terminator
```

Output Format

Print the output of each program with a single newline in between (that is, the output of programs will generally be on consecutive lines).

Sample Input

```
1
0101110107010321 0A231D0e0204170C0648656c6c6f20576f726c64000F
internal_input_unused_for_this_problem
```

Sample Output

```
Case 1: 4
Case 2: 4
Case 3: 8
```