Nineteenth Annual
University of Central Florida

High School Programming Tournament

Problems

Problem Name	Filename
Chinary Search	chinary
Purmteed Slleinpg	spelling
The Law of Signs	signs
Sleeping Astronauts	sleep
He Got the Box!	box
Vote!	vote
These Are the Voyages…	voyages
The Sixth Cents	cents
Necklaces	necklace
Duck Duck Boot!	duckboot

Call your program file: *filename*.c, *filename*.cpp, or *filename*.java
Call your input file: *filename*.in

For example, if you are solving The Law of Signs:
Call your program file: signs.c, signs.cpp or signs.java
Call your input file: signs.in

Chinary Search

Filename: `chinary`

A more appropriate name for this problem would be Chisection, but part of the title was lost in translation by our Chinese to English dictionary. Do worry not, problem translated statement perfect.

In Chinese markets, the only communication between the vendors and the customer (you) is through the LCD of a four-function calculator. You push through crowds, scooters and bicycles to find your desired item in the market. You point at it to let the vendor know that you are interested, and she types in a number into the calculator. This initial price, n, is not completely unreasonable, but one of the fun parts about being a tourist in China is haggling with the shop keepers, so you punch in a much lower number, p, than the one initially presented. The vendor laughs and mentions something about her family and how she has mouths to feed at home. Although she is not yet willing to sell you the item at your low offer, she will be; it just takes patience and persistence. She wants to get as much of your money as she possibly can, so she lowers her price by a percentage, r. But, you have been in China for a couple of days and know that the vendor will eventually reduce her price to yours, so you punch your initial offer back in the calculator to let her know that you mean business. Again, she repeats the process of lowering her current price by the percentage r, and you again type in your initial offer. This process repeats until the price she enters is less than or equal to your initial offer. How can you sleep at night knowing you haggled her down to such a low price, thus taking food out of her children's mouths?

The Problem:

You will be given three numbers: two integers representing the vendor's initial price of the item and the price that you are willing to pay. The third value will be a floating-point number indicating the percentage the vendor lowers her price each iteration. Your job is to determine how many times the Chinary Search algorithm iterates before you are satisfied.

Here is an example to illustrate how the algorithm works:

$n = 150, p = 50, r = .25$

Iteration	n	p
0	150	50
1	112	50
2	84	50
3	63	50
4	47	50

Notice that in iteration 1, the initial price 150 was dropped by 25% (37.50). This produces a new value for n, 112.50, but because prices in China rarely contain fractions of a Yuan (Chinese dollar), the vendor will always round down to set her new price when the new value contains fractions. The answer to this example would be 4.

The Input:

Each line of input will contain 3 numbers (*n, p* and *r)*, separated by one or more spaces ($0 < p < n < 2000$ and $0.001 \leq r \leq 0.999$). The input terminates when *p* equals zero.

The Output:

For each input case, print out the number of iterations required before the Chinary Search yields a result less than or equal to *p*.

Sample Input:

```
150 50 0.25
350 80 0.01
1999 2 0.20
0 0 0.0
```

Sample Output:

```
4
115
28
```

Purmteed Slleinpg

Filename: `spelling`

Psychologists have shown that we can read entire paragraphs with words that are misspelled and understand the entire paragraph. In particular, if a word has the correct first and last letter, all the letters in between can be permuted any which way, and we can still read the word properly. Consider the following paragraph:

```
You wloud tnhik taht this pgraaaprh wulod be vrey dfcluiift to
read. But inasted, you can usrednantd elaxcty waht is wtetirn
wtuoiht too mcuh sairtn. Does tihs maen you are dxyelsic, or taht
we all hvae a llitte dliysexa in us? Who kowns? But, it deos
seapk to the radnuecndy in the Eslingh lgaagune and our mnids
atiilby to cceorrt for mktiases.
```

After learning this little tidbit of trivia, your smart-aleck brother has resorted to writing all of his messages to you in this manner. When properly transformed, each word has the correct first and last letter, but the letters in between are randomly permuted. In addition to letters, the only other characters (besides spaces) his messages use are the following punctuation marks: period (`.`), comma (`,`), question mark (`?`) and exclamation point (`!`). All punctuation remains in its proper place, at the end of a word. (Though sometimes, for emphasis, he will repeat a punctuation mark at the end of a sentence several times.) Sometimes however, he accidentally changes letters such that they are no longer a properly transformed permutation of a real word.

The Problem:

Although you can read his messages, you are tired of his poor form and want to write a program that will fix his messages, or tell you that they are invalid, meaning that they contain at least one jumbled word that is not a valid transformation of a real word.

The Input:

The first line of the input will contain a single positive integer, n ($0 < n \leq 1000$), representing the number of words in the dictionary. The following n lines will contain the words from the dictionary, one word per line. All dictionary words will be entirely lowercase letters and less than 20 in length. The following line will contain a single positive integer, k ($0 < k < 100$), representing the number of messages to decipher. Each of the following k lines will contain a message each that is 80 characters or shorter, will contain at least one letter, and will not contain leading or trailing spaces. The message should be considered in a case insensitive manner, meaning that a word is valid as long as it contains the correct letters and no others, regardless of capitalization. In all original messages, the only letter in any word that will be capitalized is the first letter. Also, you are guaranteed that at most one possible word in the dictionary will match any jumbled word in any input message.

The Output:

You will produce one line of output for each of the *k* messages, each separated by a blank line. The beginning of each line of output will read:

`Message` *`m`*`:`

where *m* is replaced with the message number ($1 \leq m \leq k$).

If the message is invalid (meaning any word in the message cannot be permuted as described to match a word in the dictionary), output the following:

```
Invalid message.
```

Otherwise, correct the spelling in the message (based on the given dictionary), and output the corrected messages, with the punctuation intact. Your output should reflect the capitalization and spacing of the original message.

Sample Input:

```
12
a
an
is
can
commas
have
invalid
long
sentence
sentences
this
too
4
Tihs sceennte is lnog.
Is this a steencne?
Tihs is an invalid snteenec!
Snteecnes can hvae, cmomas, too.
```

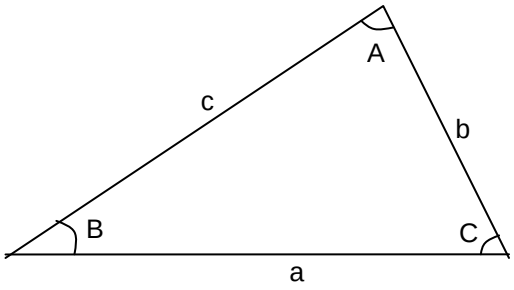
Sample Output:

```
Message 1: This sentence is long.

Message 2: Is this a sentence?

Message 3: Invalid message.

Message 4: Sentences can have, commas, too.
```

The Law of Signs

Filename: `signs`

Stan's Signs is a huge, nationwide supplier of signage for all kinds of businesses who want to advertise along the country's highways. Stan makes signs in all kinds of shapes and sizes, but his best seller has always been his triangular signs. Stan believes that the key to his success is his quality control. You see, Stan insists that his triangle signs all conform to what he calls the "Law of Signs." The Law of Signs simply states that the ratio of the length of one of the sides of the sign and the sine of the angle opposite that side must be the same for all three angle/side pairs. This may sound similar to another law that you've heard from analytic geometry, but Stan insists that it's company proprietary. To better understand the Law of Signs, here's a diagram:



The Law of Signs:

$$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$$

Stan realizes that his Law of Signs is ambiguous, in that two different angles can have the same sin value (45 degrees and 135 degrees, for example), so to account for this, he only sells acute triangular signs (all of the sign's angles are less than 90 degrees).

Unfortunately, Stan's order database recently suffered a hard drive crash, and the tape backup system failed to run that weekend, so Stan had to resort to a data recovery service to get his orders back. The service couldn't restore the data completely, so all Stan has now is partial measurements for all the signs that must be made. He's hired you to write a program to rebuild the database. Stan is clear that you're only to use the Law of Signs in this endeavor. You protest that other techniques would also be helpful in solving this problem, but Stan is worried about copyright infringement law suits from other sign companies. After a bit of convincing, Stan agrees to also let you use the triangle's angle sum property, which states that all three angles of a triangle always sum to 180 degrees.

The Problem:

Given a partial set of angle and side measurements for each sign, reconstruct the complete measurements for the sign, such that they conform to the Law of Signs and the angle sum property. Some data may have been corrupted, so if the measurements can't possibly be reconciled with each other such that the sum of the angles is within 0.01 of 180 degrees and the three terms of the Law of Signs are within 0.01 of each other, you must reject the order, and the customer must order again (a slightly annoyed customer is much better than an inferior Stan's Signs product!) Use 3.14159 as the value of Π in the recovery process. Also, in some cases there may not be enough recovered data to be able to reconstruct all the measurements using just the Law of Signs and the angle sum property. You'll need to identify these cases as well.

The Input:

There will be multiple signs. Each sign consists of one line containing six real numbers, (A, B, C, a, b, c) separated by whitespace. The first three numbers represent the angle measurements of the sign. Angles will be given in degrees (greater than 0 and less than 90), and specified in counter-clockwise order. The second three numbers will be non-zero and represent the lengths of the sides (given in inches) opposite those angles, respectively. If any of the measurements are negative, it means that that measurement is missing. Input is terminated by a line containing six negative numbers.

The Output:

For each sign, print "`Sign #`*n*`:`", where *n* is the number of the sign (starting with 1), followed by a space and the reconstructed set of three angle measurements and three side measurements, given in the same order as the input data. Print the measurements rounded to two decimal places (0.014 rounds to 0.01 and 0.015 rounds to 0.02). Separate each measurement by a single space. If a sign that conforms to both the Law of Signs and the angle sum property cannot be reconstructed, print "`Rejected!`" in place of the measurements. Similarly, if any reconstructed sign includes a non-acute angle, print "`Rejected!`" since it couldn't possibly be a Stan's Sign. If there aren't enough data to reconstruct all measurements using the techniques described, print "`Lost!`" in place of the measurements.

Sample Input:

```
60.0 -1.0 -1.0 36.0 -1.0 36.0
-1.0 65.0 -1.0 40.0 -1.0 50.0
75.0 55.0 -1.0 10.0 -1.0 10.0
-1.0 -1.0 -1.0 -1.0 -1.0 -1.0
```

Sample Output:

```
Sign #1: 60.00 60.00 60.00 36.00 36.00 36.00
Sign #2: Lost!
Sign #3: Rejected!
```

Sleeping Astronauts

Filename: `sleep`

NASA is finally sending astronauts to other planets, and you're on the team to help develop programs for them to use. Since the astronauts have been training for a while, all of the mission-critical programs (like navigation) have been completed. There's one small catch, however: not all planets in the galaxy have the same day length as Earth does. In fact, the planet they are traveling to has a 26-hour-long day. This might introduce problems with astronauts' sleep schedules. So, since regular scheduling programs can't be relied upon due to the difference in day lengths, NASA project managers have asked you to write a program to verify sleep schedules.

The Problem:

The flight surgeon is planning the sleep schedule for the astronauts, but he must run his plans through your program to verify that each astronaut is getting enough sleep during that week (seven nights) of the mission. So, given a weekly schedule, as well as the desired number of hours of sleep, your job is to see if the schedule for that astronaut is valid for one week (again, seven nights) of 26-hour-long days.

The Input:

The first line of the input file will consist of an integer, *n*, representing the number of sleep plans in the file. Each sleep plan will contain 8 lines each. The first line of the sleep plan will contain a single integer number, *x*, representing the minimum amount of total time (in hours) that that astronaut should sleep that week. The next seven lines will contain the times, in 26-hour format, for the sleep plan. The time the astronaut will go to sleep and the time the astronaut will wake up will be separated by a dash. The times will be in "`HH:MM`" format, where `HH` is the hour and `MM` is the minutes. All times will be in sequential order, meaning that the time the astronaut is scheduled to wake up will always be after the time the astronaut is scheduled to go to sleep. Also, the time the astronaut is planned to go to sleep the next night will always be after the time the astronaut is scheduled to wake up from the previous night. No astronaut will be scheduled to sleep more than 26 hours in a single sleep period. The last minute of the day will be represented as `25:59`, and one minute after that would be `00:00` (midnight). So, an elapsed time from `24:00-02:00` would be precisely four hours (2 hours before midnight, plus 2 hours after midnight.)

The Output:

If the total time the astronaut is planned to sleep during the week is enough, print out "`Astronaut #`*n* `will sleep enough!`" On the other hand, if the astronaut wouldn't be getting enough sleep, print out "`Astronaut #`*n* `needs` *x* `hours and` *y* `minutes more sleep!`", where *n* is the number of the astronaut, (starting at 1), *x* is the number of whole hours that need to be added to the sleep schedule for that astronaut, and *y* is the number of whole minutes ($0 \leq y < 60$) that need to be added to the sleep schedule for that astronaut. If

either x or y is one, you do not need to change the output to singular form ("`hours`" to "`hour`").

Sample Input:

```
2
42
23:00-05:00
22:30-04:30
20:00-05:00
25:00-05:00
25:59-00:00
25:00-09:00
22:00-05:00
40
23:00-02:00
25:00-05:00
00:00-04:00
22:15-02:15
01:00-05:00
24:00-05:00
01:00-04:45
```

Sample Output:

```
Astronaut #1 will sleep enough!
Astronaut #2 needs 4 hours and 15 minutes more sleep!
```

He Got the Box!

Filename: box

Ben has been traveling intergalactically for many years in the search of a perfect box for his portrait. His portrait is priceless, so only the best box will do. Luckily for Ben, he only has to travel in two dimensions, and two-way teleportation devices located throughout the 2D galaxy allow him to jump from one place to another instantaneously. Good thing too, because big walls may block Ben's path, and without the teleportation devices, he would be stuck without his box!

The Problem:

Ben has a map of the *m* x *n* grid in which he can travel. The map contains the locations of any teleportation devices, their destinations, the location of his precious box, and the locations of any walls within his traveling area. The problem that Ben faces is that he does not know how to get from his initial position to the box in the shortest number of steps. Your job is to move him throughout this 2D galaxy in the quickest way possible. If Ben decides that he can reach the box faster without using the teleportation devices he encounters, he does so. Also, a valid path from Ben's initial position to the box will always be present. Finally, whenever Ben reaches his beloved box, Lord Skippy screams from the clouds, "He got the Box!"

The Input:

The input will consist of several maps. The first line of a map will consist of two integer values, *m* and *n* ($0 < m \leq 20$, $0 < n \leq 20$), which indicate the number of rows and columns in Ben's map, respectively. The next *m* lines will contain exactly *n* characters, and each one of these characters has a special meaning described in the table below:

Character	Description
B	Initial position of Ben
.	Open areas Ben can travel through
W	Wall, Ben cannot move through walls
X	Box location
#	Teleportation number (described below)

The teleportation numbers are not actually represented by the # character, but as the numbers 0 through 9. If teleportation numbers are present within a map, they will always show up exactly twice. To get a better understanding of this, see the example below:

```
5 5
B....
....1
WWWWW
1....
....X
```

Ben can reach the box in 10 steps by first moving to the right 4 spaces and then down one space. He is now on teleportation device #1, and because using this device will get him to the other side of the wall, he decides to use it. After using the device, he instantaneously travels to the corresponding 1 on the other side of the wall. From there, he can move 4 more spaces to the right and down one space, making a total of 10 steps.

Exactly one `B` and `X` are guaranteed to be in each map, therefore each map will contain at least two characters. Input will terminate when *m* or *n* is zero.

The Output:

For each input case, output one line of text:

`He got the Box in` *x* `steps!`

where *x* is the minimum number of steps required for Ben to obtain the box.

Sample Input:

```
5 5
B....
....1
WWWWW
1....
....X
5 4
...B
WWW.
5XW.
WWW.
.5..
0 0
```

Sample Output:

```
He got the Box in 10 steps!
He got the Box in 7 steps!
```

Vote!

Filename: `vote`

The county of OnlyOneFemaleAndShesMarried has just had an election recently and the results have not been tallied yet. It's been two months. After much research, the citizens of OnlyOneFemaleAndShesMarried have discovered that a statewide all-female volleyball tournament is being held in the same complex that the counting is supposed to take place. After several attempts by the male citizens to "rescue" the counters failed horribly, the only female in the county decided to hire you to tally the votes so that she might know the results of the election for each district in the county.

The Problem:

Given a list of amendments and votes on those amendments determine the count for the amendment and against the amendment and then decide if the amendment passes, fails or is undecided. An amendment passes if the number of votes for it is more than the number against it.

The Input:

The first line will contain a positive integer, *n*, indicating the number of voting districts in the county.

For each district, there will be a line containing a single positive integer, *v*, which represents the number of votes given. Each of the following *v* lines will contain a line describing a vote as follows: "`Amendment x for`" or "`Amendment x against`" where *x* is a positive integer representing the number of the amendment. There will not be more than 1,000 amendments and no more than 20,000 votes in any given district.

The Output:

Begin the output for each district with a header "District *i*:" where *i* begins with 1 and increases for each district. After this print the amendments (but only those that received any votes) and the corresponding results in ascending order of Amendment number, in the following format: "`Amendment x : y for, z against : h.`" where *x* is the amendment number, *y* is the number of votes for the amendment, *z* is the number of votes against the amendment and *h* is the result of the voting for that amendment. If the amendment passed (the number of votes for was more than the number of votes against) then display "`YEA`" for *h*. If the amendment failed (the number of votes against was more than the number of votes for) then display "`NAY`" for *h*. Finally, if the voting resulted in a tie then display "`UND`" for *h*. Leave a blank line between districts.

Sample Input:

```
2
2
Amendment 1 for
Amendment 1 against
5
Amendment 1 for
Amendment 1 for
Amendment 1 against
Amendment 2 for
Amendment 2 against
```

Sample Output:

```
District 1:
Amendment 1 : 1 for, 1 against : UND.

District 2:
Amendment 1 : 2 for, 1 against : YEA.
Amendment 2 : 1 for, 1 against : UND.
```

These are the Voyages…

Filename: voyages

Earth has been attacked by an alien weapon sent by a hostile and enigmatic species, with a death toll in the millions. The starship Enterprise is the vessel sent to locate Earth's assailants and do whatever is necessary to prevent any more attacks. What the crew has learned is that the first weapon was merely a prototype. A newer weapon has been developed that could destroy our entire planet. Worse yet, the weapon has been launched and is well on its way to Earth. Now the Enterprise needs to catch up to the weapon and destroy it before it reaches Earth. You must determine how fast to travel.

There's a reason the Enterprise was sent on this mission. It is the newest, fastest, and most powerful ship in Starfleet. Its engines are designed to propel it at warp factor 5. Here's a refresher on warp drive in case you fell asleep during your course on warp field mechanics at Starfleet Academy. Warp drive allows a vessel to move at speeds faster than light by bending, or "warping," the fabric of space. The warp factor is equal to the cube root of the speed at which the ship is traveling, so warp factor 1 is the speed of light, warp factor 2 is 8 times the speed of light, warp factor 3 is 27 times the speed of light, and so on. For reasons that are only clear to warp field theoreticians, it is most efficient to travel only at integer warp factors. Because the flow of time is not constant at these speeds due to the effects of Einstein's relativity, we measure time in stardates. It takes 1000 stardate units to travel a distance of one light year at the speed of light. Even though the engines of the Enterprise are only designed for warp 5, since these are extenuating circumstances, they can be pushed to warp 6 if need be, but you'd better be ready to have the engines overhauled when you get back to Earth!

The Problem:

Compute the minimum integer warp factor necessary for the Enterprise to reach Earth before the weapon can deploy. If the Enterprise and the weapon reach Earth at the same time, this is okay because in that case there will be just barely enough time to stop the weapon before it can complete its startup sequence. Besides, it's much more dramatic to save the planet with one second left on the clock. The Enterprise is considered to arrive at the same time as the weapon if they arrive less than one tenth of a stardate unit apart.

The Input:

The first number in the input is an integer, *n,* specifying the number of scenarios in the input. Each of the following *n* lines contains a scenario consisting of the current stardate, the stardate at which the weapon will reach Earth, and the distance from the Enterprise to Earth in light years. Stardates are positive real numbers specified to exactly one digit of precision past the decimal point. The stardate at which the weapon will reach Earth will be greater than the current stardate. The distance is also a positive real number.

The Output:

For each scenario, print a heading that reads:

`Captain's log: Stardate` *`date`*

where *`date`* is the current stardate in the scenario. On the following line, if the weapon can be reached by the time it arrives at Earth, print a message indicating the minimum integral warp factor necessary to catch up in the following format:

`Warp factor` *`w- message`*

where *`w`* is that warp factor and *`message`* corresponds to the warp factor. The messages corresponding to each warp factor are in the following table:

Warp Factor	**Message**
1	Plenty of time to explore strange new worlds, too.
2	Engage.
3	Make it so.
4	We had better get boldly going.
5	Maximum warp!
6	If I give it any more, she's gonna blow!

If the ship cannot travel fast enough to reach the weapon, instead print the following message:

```
I canna change the laws of physics!
```

The output for each scenario should be followed by one blank line.

Sample Input:

```
2
3468.1 4468.1 64
5443.2 5443.3 1000
```

Sample Output:

```
Captain's log: Stardate 3468.1
Warp factor 4- We had better get boldly going.

Captain's log: Stardate 5443.2
I canna change the laws of physics!
```

The Sixth Cents

Filename: `cents`

Ali has an amazing ability. Given any fractional (decimal) amount from a purchase price, he can compute the sales tax immediately within his head. Unfortunately, as with many mathematicians he has trouble with the simplest of problems. He is unable to do it for whole dollar amounts! Luckily though, he never purchases anything over $100,000.00 in cost so we can use a simple program to help him out.

The Problem:

Given a whole dollar amount of a purchase, compute the sales tax amount based on a rate of 6%.

The Input:

The input will begin with a single positive integer, *n*, on a line by itself representing the number of purchases that need a sales tax computation. On each of the next *n* lines will be a single positive integer representing the whole dollar amount for that test purchase.

The Output:

For each purchase output the amount of sales tax charged *in cents*.

Sample Input:

```
3
1
12
99
```

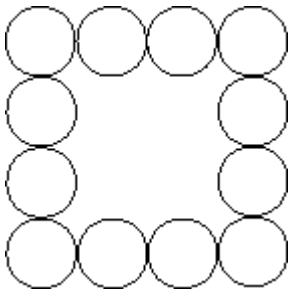
Sample Output:

```
6
72
594
```

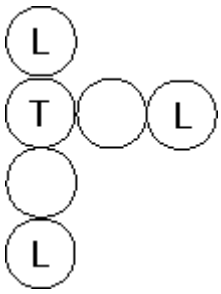
Necklaces

Filename: `necklace`

A jewelry store wants an automated way to tell whether a necklace on display is intact and displayed in a pleasing fashion. You've been contracted to develop a prototype of this system. For this initial version, a necklace will be represented as a set of circular links. In a properly-displayed necklace, each link will be tangent to exactly two other links. If a link is tanget to fewer than two other links, it is considered *loose*. If it is tangent to more than two other links, it is considered *tangled*. For example, see the two necklaces below:



A necklace with no loose or tangled links



A necklace with some loose and tangled links

The Problem:

Given a description of a necklace, write a program that will identify any loose or tangled links in a necklace.

The Input:

The input will begin with a single positive integer, n ($n \leq 10000$), on a line by itself indicating the number of necklaces to analyze. Immediately following that will be n necklace definitions. Each necklace definition will begin with a single integer, l ($0 \leq l \leq 50$), on a line by itself indicating the number of links in the necklace. On the next l lines will be a description of a necklace. Each line will contain a single link and is represented by three space-separated floating-point numbers, x, y and d ($-1000000 \leq x,y \leq 1000000$; $0.1 \leq d \leq 100000$), specifying the center (x, y) and diameter d of the link. Links may be of different sizes and may overlap; however, one link will not be completely contained within another. Links that are within 0.000001 (1e-6) unit of being tangent to each other should be considered tangent.

The Output:

For each necklace, if the necklace has no loose or tangled links, then output:

`Necklace i has no loose or tangled links.`

where *i* is the necklace number in the input (starting with 1). Otherwise, output:

`Necklace i has loose or tangled links:`

followed by the links that are loose or tangled listed in the order that they appeared in the input (links in the input are numbered beginning with 1). For a link that is loose, output the statement "`Link j is loose!`" indented two spaces where *j* is the link number. Similarly, for a link that is tangled, output "`Link j is tangled!`" again indented two spaces.

Print a blank line following the output for each necklace.

Sample Input:

```
2
12
1 1 2
1 3 2
1 5 2
1 7 2
3 1 2
3 7 2
5 1 2
5 7 2
7 1 2
7 3 2
7 5 2
7 7 2
6
1 1 2
1 3 2
1 5 2
1 7 2
3 3 2
5 3 2
```

Sample Output:

```
Necklace 1 has no loose or tangled links.

Necklace 2 has loose or tangled links:
  Link 1 is loose!
  Link 2 is tangled!
  Link 4 is loose!
  Link 6 is loose!
```

Duck Duck Boot!

Filename: `duckboot`

Jimmy's got a lot of friends. However, he likes to play this game he calls "Duck Duck Boot." It's a very simple game. He starts behind his best friend and walks around his circle of gathered friends in a clockwise direction and to each friend (beginning with his best friend) either yells "Duck" and swings, or he gives them the boot. If he gives them the boot they become agitated and then they never talk to Jimmy again (note that this also means they immediately leave the circle and are no longer a part of the game). After calling "Duck" or after booting his friend Jimmy will continue from where he is in the circle. He will not start over from his best friend's position. Now, to be fair Jimmy has decided that there must be at least one "Duck" before a boot.

The Problem:

Jimmy's mom is very concerned (she seems to recall that his birthday might be coming up) and she wants to know if Jimmy will have any friends after he finishes playing this game. She has hired you to figure out how many friends he will have after he finishes playing this game.

The Input:

The first line will contain a positive integer, g, indicating the number of games Jimmy's mom wants you to check.

For each game, there will be a line containing a single integer, f ($f > 0$), which represents the number of friends that Jimmy has before he starts playing the game (the first friend in the list is Jimmy's best friend). Each of the following f lines will contain one of his friend's names (up to 20 letters) in the order that they stand in the circle (clockwise order). All of Jimmy's friends will have a unique name and will contain only one capital letter at the beginning of his/her name (the rest of the letters will be lowercase). On the next line there is an integer, r ($0 < r \leq 2000$), and an integer, d ($0 < d \leq 2000$), separated by a single space where r represents the number of rounds that Jimmy plays with this group of friends (note that if he has no friends left he stops playing) and d represents the number of times he says "Duck" before he gives someone the boot. Jimmy is a repetitive boy and once he picks the number of times within a game that he's going to say duck before he says boot he will always stick with it.

The Output:

For each game, determine how many friends Jimmy has left after he finishes the last round of "Duck Duck Boot". Begin the output for each game with a header "Game *i*:" where *i* begins with 1 and increases for each game. After this print the list of Jimmy's remaining friends, one per line by itself (no leading or trailing spaces), in alphabetical order. If Jimmy has no friends left at the end of the game output "Jimmy has friends no more." After each game output a blank line.

Sample Input:

```
2
3
Bob
Cody
John
2 2
8
Carol
Casey
Nick
Kirsten
Ben
Bo
Billy
Heather
3 4
```

Sample Output:

```
Game 1:
Cody

Game 2:
Billy
Bo
Carol
Kirsten
Nick
```