acm International Collegiate Programming Contest

UC Programming Practice 2005
Problem Set
Universidad de Carabobo, Venezuela

August 05, 2005

This problem set should contain nine (9) problems on twenty-three (23) numbered pages. Please inform a runner immediately if something is missing from your problem set.

1 Area

Background

Being well known for its highly innovative products, Merck would definitely be a good target for industrial espionage. To protect its brand-new research and development facility the company has installed the latest system of surveillance robots patrolling the area. These robots move along the walls of the facility and report suspicious observations to the central security office. The only flaw in the system a competitor's agent could find is the fact that the robots radio their movements unencrypted. Not being able to find out more, the agent wants to use that information to calculate the exact size of the area occupied by the new facility. It is public knowledge that all the corners of the building are situated on a rectangular grid and that only straight walls are used. Figure 1 shows the course of a robot around an example area.

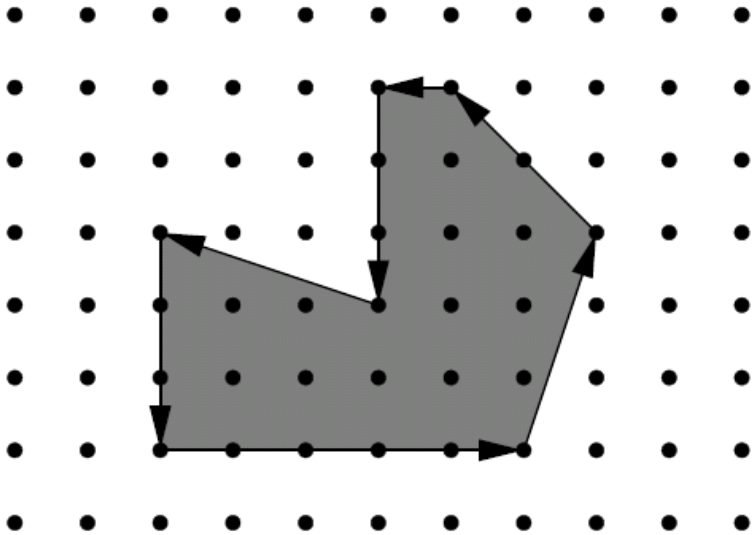

Figure 1: Example area.

Problem

You are hired to write a program that calculates the area occupied by the new facility from the movements of a robot along its walls. You can assume that this area is a polygon with corners on a rectangular grid. However, your boss insists that you use a formula he is so proud to have found somewhere. The formula relates the number I of grid points inside the polygon, the number E of grid points on the edges, and the total area A of the polygon. Unfortunately, you have lost the sheet on which he had written down that simple formula for you, so your first task is to find the formula yourself.

Input

The first line contains the number of scenarios.

For each scenario, you are given the number m, $3 \leq m < 100$, of movements of the robot in the first line. The following m lines contain pairs "dx dy" of integers, separated by a single blank, satisfying $-100 \leq \mathrm{dx}, \mathrm{dy} \leq 100$ and $(\mathrm{dx}, \mathrm{dy}) \neq (0, 0)$. Such a pair means that the robot moves on to a grid point dx units to the right and dy units upwards on the grid (with respect to the current position).

You can assume that the curve along which the robot moves is closed and that it does not intersect or even touch itself except for the start and end points. The robot moves anti-clockwise around the building,

so the area to be calculated lies to the left of the curve. It is known in advance that the whole polygon would fit into a square on the grid with a side length of 100 units.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. Then print a single line containing I, E, and A, the area A rounded to one digit after the decimal point. Separate the three numbers by two single blanks. Terminate the output for the scenario with a blank line.

Sample Input

```
2
4
1 0
0 1
-1 0
0 -1
7
5 0
1 3
-2 2
-1 0
0 -3
-3 1
0 -3
```

Sample Output

```
Scenario #1:
0 4 1.0

Scenario #2:
12 16 19.0
```

2 Ambiguous Dates

Background

More than 200 companies in more than 50 countries all over the world contribute towards the success of the *Merck Group*. You can imagine that every day *Merck Group Headquarters* at Darmstadt gets loads of mail from all over the world, the layout of all the letters following the customary style of their origin. In particular, the representation of a date is often ambiguous if you do not know in what order day, month, and year are given.

For example, if you read `01-02-03`, you do not know if that represents the first of February 1903, or 2003, or if it is the third of February 1901, or 2001. It might even be the second of March 2001, or some other permutation of the three numbers. Instead of the hyphens, there could also be slashes, backslashes, dots, commas, or no delimiters at all.

Problem

You are hired to write a program that converts dates given in an unknown format to the format of the *Adjusted Calender of Merck (ACM)*. The latter specifies the number of days relative to November 4, 2001, an important day in *Merck*'s history.

Input

The first line contains the number of scenarios.

Every scenario contains a single date on a line by itself. A date consists of three parts: A day, a month, and a year given in any order, separated either by exactly two identical delimiters, or not separated by delimiters at all. Delimiters can be slashes "`/`", backslashes "`\`", hyphens "`-`", dots "`.`", or commas "`,`".

The day and month are represented by a single digit, or by two digits, the first of which can be a leading zero. Valid years are in the range `1700...2299`; either all four digits are given, or just the last two that specify the year relative to the century. In the latter case, a leading zero may be omitted.

Dates are considered illegal if no valid interpretation exists. More precisely, a date is illegal if no classification of the digits as day, month, and year results in a valid date in the range January 1, 1700, to December 31, 2299. However, you can be sure that all dates given contain 3 to 8 digits, and no other characters except for maybe the two delimiters.

Remember that February 29 is a valid date for leap-years only. A year is a leap-year if and only if either its number is divisible by four, but not by one hundred, or if its number is divisible by four hundred. So, in particular, 2000 is a leap-year, while 1700, 1800, 1900, 2100, or 2200 are not.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1.

For every scenario, print all possible interpretations of the given date in the format of the *Adjusted Calender of Merck (ACM)*, each interpretation in a single line, in ascending order and with duplicates removed. If no valid interpretation exists, print a line containing `Illegal date` instead.

Terminate the output for each scenario with a single blank line.

Sample Input

```
3
1631/02/29
2001-11-03
010203
```

Sample Output

```
Scenario #1:
Illegal date

Scenario #2:
-238
-1

Scenario #3:
-109847
-109820
-109513
-109456
-109149
-109119
-73323
-73296
-72989
-72932
-72625
-72595
-36799
-36772
-36465
-36408
-36101
-36071
-274
-247
60
117
424
454
36250
36277
36584
36641
36948
36978
72774
72801
73108
73165
73472
73502
```

3 Cog-Wheels

Background

Your little sister has got a new mechanical building kit, which includes many cog-wheels of different sizes. She starts building gears with different ratios, but soon she notices that there are some ratios which are quite difficult to realize, and some others she cannot realize at all. She would like to have a computer program that tells her what ratios can be realized and what ratios cannot. She asks you to write a program that does the job.

For example, let us assume that the kit contains cog-wheels with 6, 12, and 30 cogs. Your sister wants to realize a gear of ratio $5 : 4$. One possible solution is shown in Figure 2.

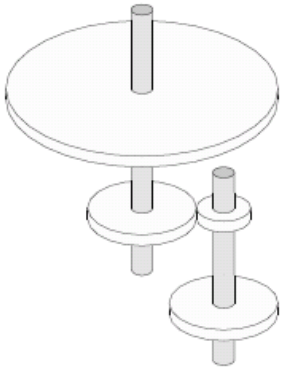

Figure 2: Combination of cog-wheels realizing a gear of $5 : 4$.

It depicts a complete gear of ratio $5 : 4$. Four wheels are used: cog-wheels of sizes 30 and 12 on the first axis, cog-wheels of sizes 6 and 12 on the second axis. The gear ratio is given by

$$\frac{30}{12} \cdot \frac{6}{12} = \frac{5}{2} \cdot \frac{1}{2} = \frac{5}{4} = 5 : 4,$$

as desired. However, a gear of ratio $1 : 6$ cannot be realized using the cog-wheels your sister has.

Problem

Given the sizes of the cog-wheels in the kit (i.e. the number of cogs they have), decide whether a given gear ratio can be built or not. You may use any finite number of cog-wheels of each size available.

Input

The input begins with a line containing the number of scenarios.

The input for each scenario starts with a description of the cog-wheels in the kit. First, there is a line containing the number n of different sizes of cog-wheels ($1 \leq n \leq 20$). The next line contains n numbers $c_1 \ldots c_n$, separated by single blanks. These denote the n different sizes of the cog-wheels in the kit, with $5 \leq c_i \leq 100$ for $i = 1, \ldots, n$. You may assume that there is a cog-wheel of smallest size $c = \min\{c_1, \ldots, c_n\}$ in the kit such that all sizes $c_1, \ldots, c_n$ are multiples of c.

The line describing the available cog-wheels is followed by the list of gear ratios to be realized. It starts with a line containing the number m of ratios. The next m lines each contain two integers a and b, separated by a single blank. They denote the ratio $a : b$, with $1 \leq a, b \leq 10000$.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. Then print the results for all the gear ratios given in that scenario. For each gear ratio $a : b$, print a line containing either

```
Gear ratio a:b can be realized.
```

or

```
Gear ratio a:b cannot be realized.
```

Terminate the output of each scenario with a blank line.

Sample Input

```
2
3
6 12 30
2
5 4
1 6
1
42
2
13 13
42 1
```

Sample Output

```
Scenario #1:
Gear ratio 5:4 can be realized.
Gear ratio 1:6 cannot be realized.

Scenario #2:
Gear ratio 13:13 can be realized.
Gear ratio 42:1 cannot be realized.
```

4 Cube

Background

After many years of development, Merck has finally discovered a simple model which helps with computer aided drug design. Different chemical substances are viewed as two-dimensional "puzzle" pieces which may be combined in a three-dimensional way, forming more complex structures of atoms. Special combinations of these pieces have very special chemical properties. These combinations which are known as *ACM* (*Anachronistic Cube Molecules*) result from combining six individual puzzle pieces along their edges to form a complete cube (see Figure 3).

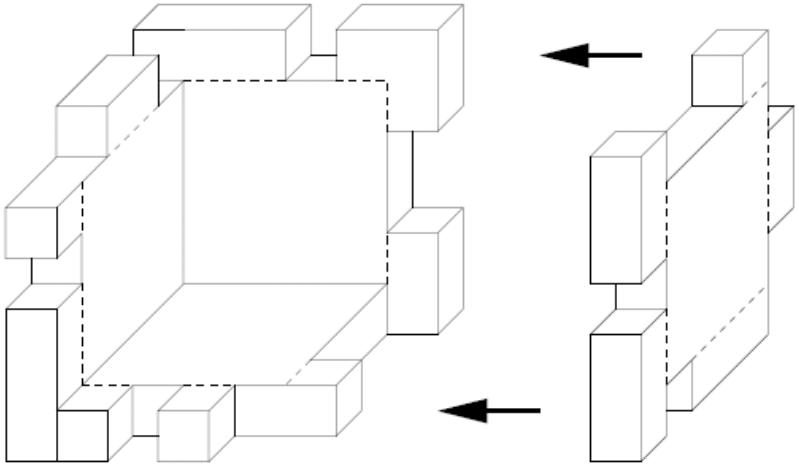

Figure 3: Construction of a cube.

Problem

Given six pieces, decide whether they can be put together to form a cube of dimension $6 \times 6 \times 6$, with no holes visible from the outside. Each piece can be thought of as being cut from a plate of wood of dimension $6 \times 6 \times 1$, with the $4 \times 4 \times 1$ plate in the centre untouched. A piece can be used the one or the other way around, i.e., there is no distinction between the inside and the outside.

Input

The first line contains the number of scenarios.

In each scenario, you are first given six lines with a graphical representation of the six pieces. In that representation, "X" symbolises solid wood, a dot "`.`" stands for a $1 \times 1 \times 1$ piece of wood that was cut out. A column with "`!`" stands on the right of each piece to separate them from each other. There is a blank line following every scenario.

It is not necessary that the pieces can really be cut out of wood or that they form a stable cube, as you can see in the sample input, second example.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. In the next line print "`Yes`" or "`No`" depending on whether the given pieces can be put together to form a cube. Terminate the output for the scenario with a blank line.

Sample Input

```
3
XX..XX!.X..X.!XX..XX!.X..X.!..XX..!..XX..!
XXXXXX!.XXXX.!XXXXXX!.XXXX.!.XXXX.!.XXXX.!
.XXXX.!XXXXXX!.XXXX.!XXXXXX!XXXXXX!XXXXXX!
.XXXX.!XXXXXX!.XXXX.!XXXXXX!XXXXXX!XXXXXX!
XXXXXX!.XXXX.!XXXXXX!.XXXX.!.XXXX.!.XXXX.!
XX..XX!.X..X.!XX..XX!.X..X.!..XX..!..XX..!

.XXXX.!.XXXX.!.XXXX.!.XXXX.!XXXXXX!X....X!
XXXXXX!.XXXX.!XXXXXX!.XXXX.!XXXXXX!.XXXX.!
XXXXXX!.XXXX.!XXXXXX!.XXXX.!XXXXXX!.XXXX.!
XXXXXX!.XXXX.!XXXXXX!.XXXX.!XXXXXX!.XXXX.!
XXXXXX!.XXXX.!XXXXXX!.XXXX.!XXXXXX!.XXXX.!
......!......!......!......!XXXXXX!X....X!

......!......!......!......!......!......!
.XXXX.!.XXXX.!.XXXX.!.XXXX.!.XXXX.!.XXXX.!
.XXXX.!.XXXX.!.XXXX.!.XXXX.!.XXXX.!.XXXX.!
.XXXX.!.XXXX.!.XXXX.!.XXXX.!.XXXX.!.XXXX.!
.XXXX.!.XXXX.!.XXXX.!.XXXX.!.XXXX.!.XXXX.!
......!......!......!......!......!......!
```

Sample Output

```
Scenario #1:
Yes

Scenario #2:
Yes

Scenario #3:
No
```

5 Enigma

Background

During the Second World War, the German military forces mainly used one special machine to secure their communication: the *Enigma* (see Figure 4). Breaking the Enigma cipher is one of the main success stories of Allied cryptanalysis and the triumph was mainly attributed to the emergence of digital computation and the genius of the people working at Bletchley Park, the secret cryptanalysis headquarters in England. The reason for this is that, while Enigma is certainly secure against pen and paper attacks, it is quite easily breakable using digital computers.

Figure 4: An Enigma machine (picture source: `http://www.nsa.gov/museum/enigma.html`).

The Enigma was a rotor machine, a cipher method which was popular at that time. A rotor is an insulated disk on which electrical contacts, one for each letter of the alphabet, are placed uniformly around the periphery and on each side. An internal conduction path through the insulating material connects contacts in pairs, one on each side of the disk. An electric current entering on one side travels on an internal path through the rotor cross-section, emerging at one of the contacts on the other side (see Figure 5 for a 3D visualisation of two rotors). Figure 6 shows a schematic side view of the complete rotor system. It shows that the Enigma has three rotors π_0, π_1 and π_2 plus an additional reflecting rotor π_R.

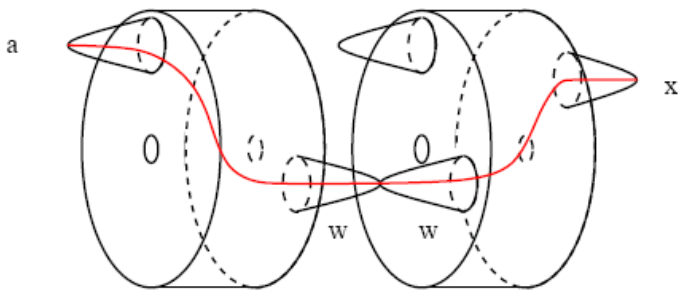

Figure 5: 3D view of two rotors.

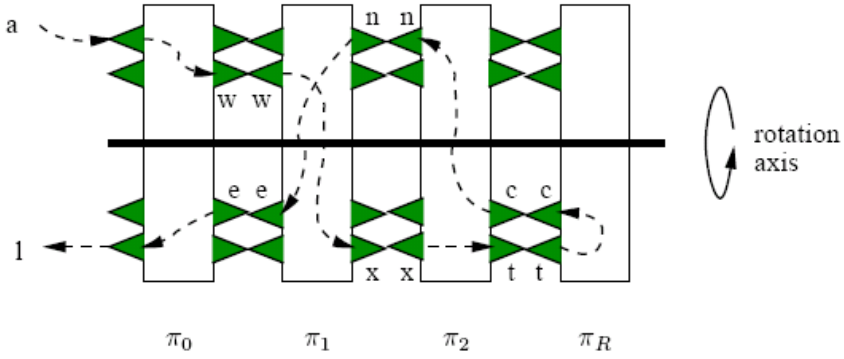

Figure 6: Side view of the Enigma's rotor system.

The input to Enigma is a stream of alphabetic characters without blanks. Every character is subject to the following steps:

1. The plaintext is subject to an initial permutation IP which is implemented by a *plugboard*.
2. The character resulting from step 1 is sent through the three rotors π_0, π_1 and π_2.
3. The resulting character is then sent through the reflecting rotor π_R.
4. The character from step 3 is passed back through the rotors π_2, π_1, and π_0 (i.e., in the opposite direction).
5. The character from step 4 is subject to the inverse IP^{-1} of the initial permutation IP.

The interesting point about the use of rotors is that after processing each character, every rotor might be rotated by a certain angle (i.e., a certain amount of letters) before processing the next character. With the Enigma, rotor π_0 is rotated by one in anti-clockwise direction with every new character. When π_0 has finished one round (i.e., after processing 26 characters), rotor π_1 moves by one character. Similarly, rotor π_2 is rotated by one character when π_1 has finished one revolution, and the reflecting rotor π_R moves when π_2 has finished its rotation. Obviously, π_R is the slowest of the four rotors.

The process described above can be used both for encryption and decryption, provided that the permutation π_R implemented by the reflecting rotor is an involution. That means $\pi_R = \pi_R^{-1}$, or, equivalently, $\xi = \pi_R(\zeta)$ whenever $\zeta = \pi_R(\xi)$. You may assume that this condition holds.

The secret key of the Enigma consists of (1) the rotors π_0, π_1, π_2, and π_R, (2) the plugboard permutation IP, and (3) the initial rotational displacements k_0, k_1, k_2, k_R of π_0, π_1, π_2, and π_R (see below). The rotors were changed infrequently and were selected from a set of four possible rotors in the Wehrmacht model.

Problem

You are time-warped to Bletchley Park together with your laptop and should help to decipher some messages which have been intercepted over the day. You are given the entire ciphertext, parts of the plaintext, and parts of the Enigma key. Your task is to determine the correct key and finally complete the plaintext by decoding the ciphertext.

Input

The first line contains the number of scenarios.

Each scenario begins with the secret key of the Enigma. The secret key is specified by 6 lines. The first four lines contain a specification of the rotors π_0, π_1, π_2 and π_R as a sequence of lowercase alphabetic characters. Character i ($1 \leq i \leq 26$) gives the mapping of the i-th character of the alphabet (e.g., "`bha...`" means that "`a`" is mapped to "`b`", "`b`" is mapped to "`h`", "`c`" is mapped to "`a`" etc.). Physically, the sequence of characters is given in clockwise direction looking from the front of the rotor stack $\pi_0, \ldots, \pi_R$.

After the rotors follows a similar line giving the plugboard permutation IP. Finally, the sixth line of the key gives the initial displacement k_0, k_1, k_2, k_R of the four rotors π_0, π_1, π_2, and π_R as a string of four characters where "`a`" means that the rotor is in its original position (as defined by the rotor specification above), "`b`" means that it is rotated by one position in the usual way etc. For example, "`dgaa`" means that rotor π_0 has initial displacement 3, π_1 has 6, and π_2, π_R are both in their original position.

After the key follow two lines, each containing at least 1 and at most 80 lowercase letters, and no other characters. The first line contains the plaintext while the second line contains the ciphertext.

The plaintext and any part of the key may be *incomplete*, i.e., some positions in the strings may be question marks "`?`". The number of question marks in the input will be at most 3.

Output

The output for each scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. In the next line you are to output the completed, decrypted plaintext. You can assume that a solution exists and that it is unique. Terminate the output for each scenario with a blank line.

Sample Input

```
2
wfbtiznuvcqejpokshxgmadyrl
hmrgnqpkjcaivwluebfzsyxtdo
druahlbfzvgmwckxpiqysontje
owtvskypjifmluahrqecndbzgx
?bcdefghijklmnopqrstuvwxyz
aaaa
manyorganizationsrelyoncom??ters
grsuztldsznkwnerdpfbovvqnobkyiqn
oqzunvhtxwryfebicmjpklsgda
zupogrskynxtwdfqvbliejcmha
kzvlyjuodmscewxtfbphriqgna
gbcnylaztwkfmdspqvoiurjxeh
rfyhkxbuvplgtqmdiewjosznca
dmeo
???
ave
```

Sample Output

```
Scenario #1:
manyorganizationsrelyoncomputers

Scenario #2:
acm
```

6 Gridland

Background

For years, computer scientists have been trying to find efficient solutions to different computing problems. For some of them efficient algorithms are already available, these are the "easy" problems like sorting, evaluating a polynomial or finding the shortest path in a graph. For the "hard" ones only exponential-time algorithms are known. The *traveling-salesman problem* belongs to this latter group. Given a set of N towns and roads between these towns, the problem is to compute the shortest path allowing a salesman to visit each of the towns once and only once and return to the starting point.

Problem

The president of Gridland has hired you to design a program that calculates the length of the shortest traveling-salesman tour for the towns in the country. In Gridland, there is one town at each of the points of a rectangular grid. Roads run from every town in the directions North, Northwest, West, Southwest, South, Southeast, East, and Northeast, provided that there is a neighbouring town in that direction. The distance between neighbouring towns in directions North–South or East–West is 1 unit. The length of the roads is measured by the Euclidean distance. For example, Figure 7 shows 2×3-Gridland, i.e., a rectangular grid of dimensions 2 by 3. In 2×3-Gridland, the shortest tour has length 6.

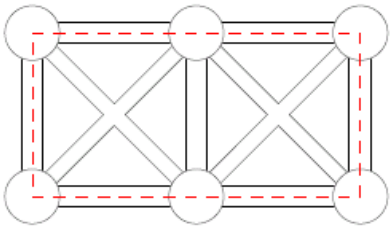

Figure 7: A traveling-salesman tour in 2×3-Gridland.

Input

The first line contains the number of scenarios.

For each scenario, the grid dimensions m and n will be given as two integer numbers in a single line, separated by a single blank, satisfying $1 < m < 50$ and $1 < n < 50$.

Output

The output for each scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. In the next line, print the length of the shortest traveling-salesman tour rounded to two decimal digits. The output for every scenario ends with a blank line.

Sample Input

```
2
2 2
2 3
```

Sample Output

```
Scenario #1:
4.00

Scenario #2:
6.00
```

7 T9

Background

A while ago it was quite cumbersome to create a message for the Short Message Service (SMS) on a mobile phone. This was because you only have nine keys and the alphabet has more than nine letters, so most characters could only be entered by pressing one key several times. For example, if you wanted to type "`hello`" you had to press key 4 twice, key 3 twice, key 5 three times, again key 5 three times, and finally key 6 three times. This procedure is very tedious and keeps many people from using the Short Message Service.

This led manufacturers of mobile phones to try and find an easier way to enter text on a mobile phone. The solution they developed is called *T9 text input*. The "9" in the name means that you can enter almost arbitrary words with just nine keys and without pressing them more than once per character. The idea of the solution is that you simply start typing the keys without repetition, and the software uses a built-in dictionary to look for the "most probable" word matching the input. For example, to enter "`hello`" you simply press keys 4, 3, 5, 5, and 6 once. Of course, this could also be the input for the word "`gdjjm`", but since this is no sensible English word, it can safely be ignored. By ruling out all other "improbable" solutions and only taking proper English words into account, this method can speed up writing of short messages considerably. Of course, if the word is not in the dictionary (like a name) then it has to be typed in manually using key repetition again.

Figure 8: The Number-keys of a mobile phone.

More precisely, with every character typed, the phone will show the most probable combination of characters it has found up to that point. Let us assume that the phone knows about the words "`idea`" and "`hello`", with "`idea`" occurring more often. Pressing the keys 4, 3, 5, 5, and 6, one after the other, the phone offers you "`i`", "`id`", then switches to "`hel`", "`hell`", and finally shows "`hello`".

Problem

Write an implementation of the *T9 text input* which offers the most probable character combination after every keystroke. The probability of a character combination is defined to be the sum of the probabilities of all words in the dictionary that begin with this character combination. For example, if the dictionary contains three words "`hell`", "`hello`", and "`hellfire`", the probability of the character combination "`hell`" is the sum of the probabilities of these words. If some combinations have the same probability, your program is to select the first one in alphabetic order. The user should also be able to type the beginning of words. For example, if the word "`hello`" is in the dictionary, the user can also enter the word "`he`" by pressing the keys 4 and 3 even if this word is not listed in the dictionary.

Input

The first line contains the number of scenarios.

Each scenario begins with a line containing the number w of distinct words in the dictionary ($0 \leq w \leq 1000$). These words are given in the next w lines in ascending alphabetic order. Every line starts with the word which is a sequence of lowercase letters from the alphabet without whitespace, followed by a space and an integer p, $1 \leq p \leq 100$, representing the probability of that word. No word will contain more than 100 letters.

Following the dictionary, there is a line containing a single integer m. Next follow m lines, each consisting of a sequence of at most 100 decimal digits 2–9, followed by a single 1 meaning "next word".

Output

The output for each scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1.

For every number sequence s of the scenario, print one line for every keystroke stored in s, except for the 1 at the end. In this line, print the most probable word prefix defined by the probabilities in the dictionary and the T9 selection rules explained above. Whenever none of the words in the dictionary match the given number sequence, print "`MANUALLY`" instead of a prefix.

Terminate the output for every number sequence with a blank line, and print an additional blank line at the end of every scenario.

Sample Input

```
2
5
hell 3
hello 4
idea 8
next 8
super 3
2
435561
43321
7
another 5
contest 6
follow 3
give 13
integer 6
new 14
program 4
5
77647261
6391
4681
26684371
77771
```

Sample Output

```
Scenario #1:
i
id
hel
hell
hello

i
id
ide
idea


Scenario #2:
p
pr
pro
prog
progr
progra
program

n
ne
new

g
in
int

c
co
con
cont
anoth
anothe
another

p
pr
MANUALLY
MANUALLY
```

8 Number Game

Background

Christiane and Matthias are playing a new game, the *Number Game*. The rules of the Number Game are:

Christian and Matthias take turns in choosing integer numbers greater than or equal to 2. The following rules restrict the set of numbers which may be chosen:

R1 A number which has already been chosen by one of the players or a multiple of such a number cannot be chosen. (A number z is a *multiple* of a number y if z can be written as $y \cdot x$ and x is a positive integer.)

R2 A sum of two such multiples cannot be chosen either.

R3 For simplicity, a number which is greater than 20 cannot be chosen either. This enables a lot more NPCs (Non-Personal-Computers) to play this game.

The player who cannot choose any number anymore looses the Number Game.

Here is an example: Matthias starts by choosing 4. Then Christiane is not allowed to choose 4, 8, 12, etc. Let us assume her move is 3. Now, the numbers 3, 6, 9, etc. are excluded, too; furthermore, numbers like: $7 = 3 + 4, 10 = 2 \cdot 3 + 4, 11 = 3 + 2 \cdot 4, 13 = 3 \cdot 3 + 4, \ldots$ are not available. So, in fact, the only numbers left are 2 and 5. Matthias now says 2. Since $5 = 2 + 3$ is now forbidden, too, he wins because there is no number for Christiane's move left.

Your task is to write a program which will help to play the Number Game. In general, i.e., without rule R3, this game may go on forever. However, with rule R3, it is possible to write a program that finds a strategy to win the game.

Problem

Given a game situation (a list of numbers which are not yet forbidden), your program should output all *winning moves*. A winning move is a move by which the player whose turn it is can force a win no matter what the other player will do. Now we define these terms more formally:

- A *loosing position* is a position in which either
  1. all numbers are forbidden, or
  2. no winning move exists.
- A *winning position* is a position in which a winning move exists.
- A *winning move* is a move after which the position is a loosing position.

Input

The first line contains the number of scenarios.

The input for each scenario describes a game position. It begins with a line containing the number a, $0 \leq a < 20$ of numbers which are still available. Next follows a single line with the a numbers still available, separated by single blanks.

You may assume that all game positions in the input could really occur in the Number Game (for example, if 3 is not in the list of numbers available, 6 will not be, either).

Output

The output for each scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. In the next line either print "`There is no winning move.`" if this is true for the position of the current scenario, or "`The winning moves are:` $w_1\ w_2 \ldots w_k$`.`" where the w_i are all the winning moves, in ascending order, separated by single blanks. The output for each scenario should be followed by a blank line.

Sample Input

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

Sample Output

```
Scenario #1:
The winning moves are: 2.

Scenario #2:
There is no winning move.
```

9 Signal Box

Background

On its trip, a train has to pass a lot of points (American English: switches) and signals. The train's track depends on the status of points and signals. The responsible operator on the signal box does not handle them separately, but tells the signal box the start and destination signal of the train's journey. The box then determines the correct status of points and signals and brings them into the right position.

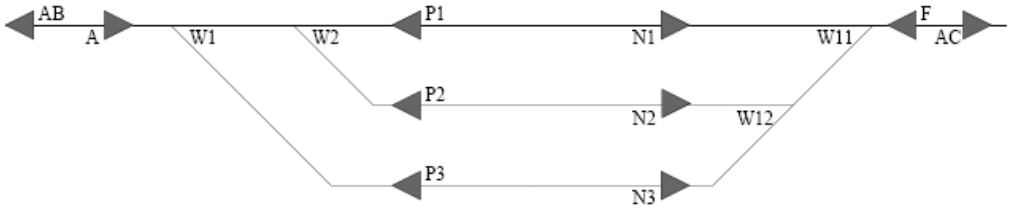

Figure 9: Schematic view of points and signals on a sample train track.

Figure 9 shows a sample scenario in which railway tracks are shown as solid lines and signals are drawn as triangles (this is also the first scenario of the sample input). Signals have a sense of direction: they are only valid for the direction in which the triangle points (e.g., signal A is valid for trains running from left to right, see also Figure 10). Points are located where railway tracks meet (e.g., at points W1, W2, etc.). Points have a *front* side (i.e., the side from which a train can take alternative directions) and a *back* side and can be in two positions, named + and −. If a train comes from the front side, it leaves the point at the + or − leg, dependent on the point's position (see Figure 11). If the train comes from one of the the back legs, it leaves it at the front leg. Even then the point has to be put into the right state, otherwise it gets damaged!

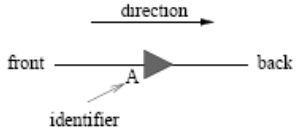

Figure 10: Signal valid for trains running from left to right

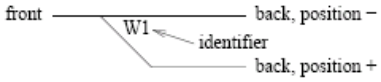

Figure 11: Point with two possible positions

Problem

Your task is to implement an automatic signal box, i.e., write a program which finds the correct position of points and signals for a given start and destination. The signal box should follow these rules:

- A journey can only start and end at a signal. Both signals have to be in the same direction!
- During a journey a train must not change its direction.
- The journey consists of a sequence of signal and point settings. A signal is only taken into account for the journey if it has the right direction. A point along the way is always taken into account.

- If there is more than one possible track from the start signal to the destination signal, the correct one is determined by the following scheme:
  - Consider a set of *path selection rules*. These are given as a triple (x, y, z) of point identifiers x and y, and a position z. A selection rule has the following meaning:

    If there are alternative paths starting at point x and ending at point y where x is approached from the front and y from the back, then consider only paths in which x is in position z (z is either $+$ or $-$).
  - If no such rule exists for a given point x, the $-$ position must be chosen.

The sample in- and output demonstrate the application of the rules. Furthermore, you can make the following assumptions:

- The track plan is acyclic.
- Within a path, each element is only used once or not at all.
- If for a given point x several rules (x, y, z) exist, they will agree on the position to be chosen.

Input

The first line contains the number of scenarios.

In the first line of the input for every scenario, you are given two signal identifiers for the departure and the destination, separated by a single blank. The following line contains the number n of elements (points and signals) in the track plan. You can assume $1 \leq n \leq 200$ and that each element has a unique identifier of at most 20 alphanumeric characters. The identifier "`XXX`" is given to track ends.

There are signal and point elements, given in the following format:

- Points are specified by a line "`W I F M P`", where `W` stands for "Weiche" (German for point), `I` is the identifier of the point, `F` identifies the front element of the point, and `M` and `P` give the identifiers of the back elements of the point depending on whether it is in *m*inus or *p*lus position.
- Signals are specified by a line "`S I F B`", where `S` stands for "Signal" (German for signal), `I` is the identifier of the signal, and `F` and `B` give the identifiers of the front and back elements of the signal. The direction for which the signal is valid is from front to back.

The following line contains the number p, $0 \leq p \leq 100$, of path selection rules, followed by another p lines of the rules themselves. A rule is of the form "`FW X Y Z`" where `FW` is the identifier of "Fahrstraßenwahl-Regel" (German for path selection rule), `X`, `Y` and `Z` are the elements of the rule as explained above.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1.

For every scenario print out the elements on the path from departure to destination in the order they are passed by the train. However, print the signals first, followed by the points. Every element of the path must be on a line by itself. Elements of the path are signal and point identifiers (the first and the last signal identifiers must also be printed). For every point you should also give the correct position of the point as either $+$ or $-$ on the same line, separated from the point identifier by a single blank. If there is no possible path, print "`NOT POSSIBLE`".

Terminate each scenario by a blank line.

Sample Input

```
4
A AC
14
S AB A XXX
S A AB W1
W W1 A W2 P3
W W2 W1 P1 P2
S P1 N1 W2
S N1 P1 W11
W W11 F N1 W12
S F AC W11
S AC F XXX
S P2 N2 W2
S N2 P2 W12
W W12 W11 N3 N2
S P3 N3 W1
S N3 P3 W12
2
FW W1 W11 +
FW W11 W1 -
S1 S2
2
S S1 S2 XXX
S S2 S1 XXX
0
S1 S4
6
S S1 XXX W1
S S2 W1 XXX
S S3 XXX W2
S S4 W2 XXX
W W1 S1 S2 W2
W W2 S4 W1 S3
0
S1 S2
8
S S1 XXX W1
S S2 W4 XXX
S S3 W1 W2
S S4 W3 W4
W W1 S1 W2 S3
W W2 W3 W1 S3
W W3 W2 W4 S4
W W4 S2 W3 S4
1
FW W1 W2 +
```

Sample Output

```
Scenario #1:
A
N3
AC
W1 +
W12 -
W11 +

Scenario #2:
NOT POSSIBLE

Scenario #3:
S1
S4
W1 +
W2 -

Scenario #4:
S1
S3
S2
W1 +
W2 +
W3 -
W4 -
```