2013 University of Virginia High School Programming Contest

Welcome to the 2013 University of Virginia High School Programming Contest. Before you start the contest, please be aware of the following notes:

The Contest

1. There are nine (9) problem(s) in the packet, using letters A-I. These problems are approximately sorted by difficulty, with the easiest problems first. As a team's solution is judged correct, the team will be awarded a balloon. The balloon colors are as follows:

Problem	Problem Name	Baloon Color
A	Gradabase	Orange
B	Jenga	Blue
C	More Dice	Purple
D	Morse Code	Pink
E	Tic Tac Toe For the Win	Gray
F	Sudoku Solution Checker	Green
G	Robot In A Room	Red
H	Excellent Adventure	Yellow
I	Even More Dice	Black

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 is issued during the contest, it will be broadcast to all teams.

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.

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). 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.

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!!!

All images in this packet are from WikiMedia.

A. Gradabase

It's the end of the year for Hunt Valley Elementary. Everyone passed their classes and is moving forward a grade; update everyone's grade level in the database. Kindergarden is represented by a 0, First grade by a 1, and so on. The Hunt Valley Elementary is a K-6 school. Only print out student's grade levels who are still at Hunt Valley Elementary.

Input

The first line of input is a number, n, representing the number of test cases. Each case will start with a number, m, representing the number of entries in that test case followed by a list of m integers between 0 and 6, with each number on its own line. There will be anywhere from 1 to 500 test cases with anywhere from 1 to 500 students per case

Output

For each case, output the line "Case x:" where x is the case number, on a single line, followed by a list of integers, each on a new line, between 1 and 6. If the student has graduated from the school, do not print them.

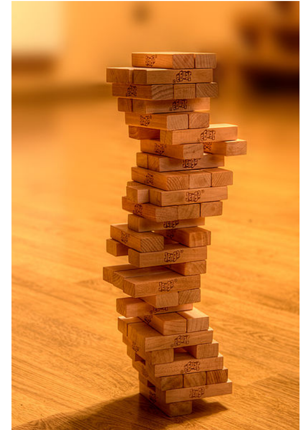
Sample Input

```
3
10
6
1
2
3
6
6
6
1
0
0
2
1
1
1
6
```

Sample Output

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

B. Jenga



You are writing a computer program that will play Jenga, but first you need to figure out what boards are legal! Given a set of possible jenga boards, determine if each board is standing or if it has fallen over. A board falls over if any two consecutive blocks in a row (not column) are both missing, including within the top row. Assume that even two consecutive missing blocks on the top row makes the tower "Fallen."

Input

The first line of input is the number of test cases that follow.

Each input case will start with a single integer representing the height of the Jenga tower. Each row in the tower appears on a single line and represents the blocks currently still in the board for that row. A '1' represents the block being present, a '0' represents a removed block.

There will be between 1 and 100 test cases, each with one jenga board of height between 1 and 20.

Output

For each case, output the line "Case x:" where x is the case number, on a single line followed by a single space and either the word "Fallen" or "Standing"

Sample Input

```
4
8
111
111
111
100
101
111
010
111
4
111
111
101
101
5
000
111
111
111
101
8
111
010
111
011
100
101
101
111
```

Sample Output

```
Case 1: Fallen
Case 2: Standing
Case 3: Fallen
Case 4: Fallen
```

C. More Dice

You roll two standard six-sided dice. Given the sum of your roll, determine the dice combinations that would result in that sum. Only list unique dice combinations: e.g. (1,6) and (6,1) are equivalent. List only one of these.

Input

The first line of input is the number of test cases that follow.

Each input case appears on a single line, and will be a single integer; the sum of a roll of two 6-sided dice.

There will be at most 1000 input cases.

Output

For each case, output the line "Case x:" where x is the case number, on a single line. Then output a list of possible dice-pairs that result in that sum, one on each line. Each dice-pair should be comma-separated and enclosed by parentheses. In each pair, the die values should be ordered from lowest to highest. The pairs should also be ordered from lowest to highest based on the lowest die.

Sample Input

```
2
7
5
```

Sample Output

```
Case 1:
(1,6)
(2,5)
(3,4)
Case 2:
(1,4)
(2,3)
```

D. Morse Code

Morse code was an early method of communication via electronic signals. Each letter and number was represented by a unique series of long and short tones, with a pause to indicate the beginning of the next character. Implement a Morse Code interpreter, using the key to the right, that translates five-letter messages (no more, no less).

A •–
B –•••
C –•–•
D –••
E •
F ••–•
G ––•
H ••••
I ••
J •–––
K –•–
L •–••
M ––
N –•
O –––
P •––•
Q ––•–
R •–•
S •••
T –
U ••–
V •••–
W •––
X –••–
Y –•––
Z ––••

Input

The first line of input is the number of test cases that follow.

Each input case appears on a single line, and will include five Morse code characters, a space-separated series of long and short tones, represented by dots and dashes.

There will be at most 1000 test cases.

Output

For each case, output the line "Case x:" where x is the case number, on a single line. Then output a single space followed by an all-caps alphanumeric representation of the message, exactly five characters in length.

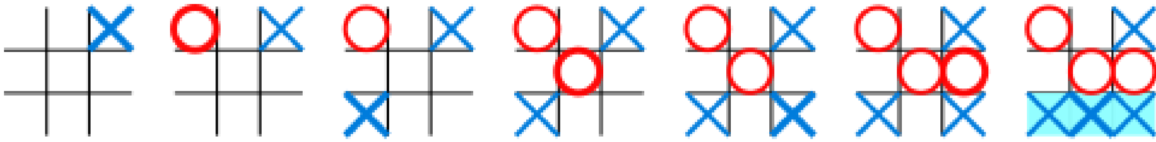
Sample Input

```
5
... ..- .--. . .-.
... .- .-.. .- -..
--.- ..- . ... ---
-.. .. ... -.-. ---
-. -.-- .-.. --- -.
```

Sample Output

```
Case 1: SUPER
Case 2: SALAD
Case 3: QUESO
Case 4: DISCO
Case 5: NYLON
```

E. Tic Tac Toe For the Win



You are playing a game of tic-tac-toe with your friend and you are about to win! Given a tic-tac-toe board that is one step away from victory, make the winning move!

Input

The first line will be the number of games that you are playing. Each game board will come in 3x3 blocks of characters, where x and o are players' markers, and a dash (-) represents an open square. Each board will be followed by a single x or o to represent which player you are. You will have only two of your markers on the board, and you are guaranteed to be able to win the game in one move.

Output

For each case, output the line "Case x:" where x is the case number, on a single line. On the next 3 lines, output the board as it will look after your winning move. Use x and o to represent players' markers and a dash (-) to represent open squares.

Sample Input

```
3
o--
-o-
xx-
x
o-x
--o
x--
x
xx-
o-o
---
o
```

Sample Output

```
Case 1:
o--
-o-
xxx
Case 2:
o-x
-xo
x--
Case 3:
xx-
ooo
---
```

F. Sudoku Solution Checker

Sudoku is a Japanese word that translates to "Single Number." It is also the name of a Japanese game that is popular in America. A Sudoku board is a 9x9 grid with numbers in it. In order to solve a Sudoku Puzzle, you must satisfy the following conditions:

3	5	7	9	6	4	2	8	1
4	6	8	1	2	3	5	7	9
9	1	2	5	8	7	4	6	3
6	3	1	7	9	5	8	4	2
7	2	4	3	1	8	6	9	5
8	9	5	2	4	6	1	3	7
1	7	6	4	5	9	3	2	8
5	8	3	6	7	2	9	1	4
2	4	9	8	3	1	7	5	6

- Each integer 1-9 must appear in each row exactly once
- Each integer 1-9 must appear in each column exactly once
- Each integer 1-9 must appear in each 3x3 square exactly once.

The puzzle to the right is a correctly-completed Sudoku puzzle.

You have recently become obsessed with Sudoku puzzles - you play them all the time! But you hate the fact that there is no easy way to verify that your puzzles are completed correctly. Write a program that can identify a correctly-completed Sudoku board.

Input

The first line will be a single integer representing the number of test cases. Each test-case will be 9 rows, each containing 9 space-separated integers between 1 and 9. There will be one blank line between each test case.

There will be at most 100 test cases.

Output

For each case, output the line "Case x:" where x is the case number, on a single line followed by a single space. Then print "CORRECT" if the puzzle is solved correctly, and "INCORRECT" if it is solved incorrectly.

Sample Input

```
2
1 2 3 5 6 7 4 8 9
3 4 5 6 1 2 4 7 8
7 5 8 3 4 2 1 9 6
1 2 3 5 6 7 4 8 9
3 4 5 6 1 2 4 7 8
7 5 8 3 4 2 1 9 6
1 2 3 5 6 7 4 8 9
3 4 5 6 1 2 4 7 8
7 5 8 3 4 2 1 9 6

3 5 7 9 6 4 2 8 1
4 6 8 1 2 3 5 7 9
9 1 2 5 8 7 4 6 3
6 3 1 7 9 5 8 4 2
7 2 4 3 1 8 6 9 5
8 9 5 2 4 6 1 3 7
1 7 6 4 5 9 3 2 8
5 8 3 6 7 2 9 1 4
2 4 9 8 3 1 7 5 6
```

Sample Output

```
Case 1: INCORRECT
Case 2: CORRECT
```

G. Robot In A Room

You just bought a new robotic vacuum cleaner, and you love to watch it explore your room. Through careful observation, you have determined that it follows the following navigation rules:

- It moves one step per minute.

- It has a limited battery that allows it to move n steps.

- It can plug into any outlet. If it passes an outlet and has less than or equal to half of its battery left, it will stop to charge. A robot can charge from any outlet that it is directly adjacent to (not diagonal).

- It always charges until it is full, and charging takes one minute per unit of battery charged.

- When it runs into a wall, it turns right 90 degrees (which takes one minute and one unit of battery).

Unfortunately, you just spilled your drink on the carpet. Given the current position of the robot and the position of your spill, determine how many steps it will take your robot to clean your mess. The robot cleans a mess as soon as it enters a space, so cleaning takes no additional power.

Input

The first line will be a single integer representing the number of test-cases. Each test case will start with two numbers separated by a space, n and d. n represents the size of your room, which is an n-by-n unit square. d is the total distance that your robot can travel on a single charge. n will be between 5 and 20, d will be between 15 and 30.

The following n lines will each contain n characters with the following meanings:

- represents open space

x represents an obstacle (such as a wall or a chair)

m represents your mess

r is the starting position of your robot (it will always start facing right with full charge). It will never start in the wall.

p represents a power outlet

There will be at most 500 test cases.

Output

For each case, output the line "Case x:" where x is the case number, on a single line followed by a single space and the number of minutes that it will take your robot to reach your mess. If the robot will never reach your mess, output the word 'NEVER'.

Sample Input

```
2
5 10
xxpxx
x---x
x-m-x
xr--x
xxxxx
10 30
xxxxxxxxxx
x-x------x
x--------x
xr-xx----x
x--xx----p
xm-------x
xx------xx
xxxx-----x
x--------x
xxxxxxxxxx
```

Sample Output

```
Case 1: NEVER
Case 2: 51
```

H. Excellent Adventure

You are on a road trip around Virginia with your friends and you only have a limited amount of gas. There are a series of optional excursions to roadside attractions. Find the route from your starting city to your destination that maximizes your units of fun without running out of gas!

Although you may pass through a city multiple times on one road trip, you only stop and have fun once in each city.

Input

The first line of input will be a single integer, c, the number of cities you can visit.

The following c lines will each be a city name followed by a single space, followed by an integer representing its fun value.

The next line will be a number m, representing the number of roads between cities that you can travel. The next m lines each represent one road. Each line includes a city name followed by a single space and another city name followed by a single space and the length of the road between the cities in miles (assume all roads are of integer length). All roads are two directions. The next line is a single integer p, representing the number of test cases. The following p lines will include the name of your starting city, a single space, the name of your destination city, a single space, and the total distance in miles that you are allowed to drive.

City names will include no whitespace (eg. "VirginiaBeach" is an acceptable city name, "Virginia Beach" is not).

Output

For each case, output the line "Case x:" where x is the case number, on a single line followed by a space and the optimal path in order. The path should be expressed as each city that you will drive through in order, space delimited. Then, on the same line, output the total fun value from the cities that you will visit. The fun value includes the fun values of your starting and ending cities.

If there is no valid path from the start to destination, instead of a route and value, print "Not possible".

Note that despite any wraparound in the printout, any route is on a single line with its fun value. If there is any uncertainty in interpretation, consult your digital copy of the output.

Sample Input

```
10
VirginiaBeach 4
Richmond 5
Charlottesville 10
Blacksburg -5
Roanoke 3
Fredericksburg 3
Danville 8
Harrisonburg 3
Lynchburg 7
Arlington 5
7
VirginiaBeach Richmond 90
Richmond Charlottesville 72
Charlottesville Lynchburg 65
Richmond Fredericksburg 75
Fredericksburg Arlington 40
Lynchburg Roanoke 30
Roanoke Blacksburg 20
3
VirginiaBeach Charlottesville
   400
Charlottesville Lynchburg 75
VirginiaBeach Richmond 10
```

Sample Output

```
Case 1: 29
VirginiaBeach
Richmond
Charlottesville
Lynchburg
Roanoke
Lynchburg
Charlottesville
Case 2: 17
Charlottesville
Lynchburg
Case 3: Not possible
```

I. Even More Dice

At Carlisle's Casino there's a new dice game that everyone is talking about. Given n dice, each with m sides, the player has to roll the sum randomly chosen by the dealer. For each number given by the dealer, compute the possible rolls that would result in that sum.

Input

The first line of input is a single integer, c, representing the number of test cases. the next line contains three space-separated integers, representing the n, the number of dice, and m, the number of sides that each die has, and finally s, the sum that must be rolled.

Output

For each case, output the line "Case x:" where x is the case number, on a single line. Then list each of the possible dice combinations that result in that sum, one on each line. Each dice value should be comma-separated and enclosed by parentheses. All dice must be used in each set. Each ordered set should be least to greatest when reading from left to right, and one permutation of each combination should be shown. For example, (1,2,3) is correct where as (2,3,1) (2,1,3) (3,1,2) (3,2,1) (1,3,2) are incorrect.

Sample Input

```
1
3 8 12
```

Sample Output

```
Case 1:
(1,3,8)
(1,4,7)
(1,5,6)
(2,2,8)
(2,3,7)
(2,4,6)
(2,5,5)
(3,3,6)
(3,4,5)
(4,4,4)
```