University Interscholastic League

Computer Science Competition

2004 Regional Programming Set

I. General Notes

1. Do the problems in any order you like. They do not have to be done in order from 1 to 10.

2. All problems have a value of 6 points.

3. There is no extraneous input. All input is exactly as specified in the problem. Unless specified by the problem, integer inputs will not have leading zeros. Unless otherwise specified, your program should read to the end of file.

4. Your program should not print extraneous output. Follow the form exactly as given in the problem.

II. Point Values and Names of Problems

Number	Name	Point Value
Problem 1	This Is Correct!	6
Problem 2	Test Scores	6
Problem 3	List the Primes	6
Problem 4	Arena	6
Problem 5	Dice Golf	6
Problem 6	Matrix	6
Problem 7	Matrix Reloaded	6
Problem 8	PacMan	6
Problem 9	drawkcaBsay iPgay itaLnay Day	6
Problem 10	Roman Numeral Translator	6
Total		**60**

Problem 1		6 Points

This Is Correct!

Program Name: correct.java **Input File: [none]**

This problem requires no input. It also requires no calculations. Simply output the message, "This Is Correct!" to get credit for this problem.

Input
There is no input.

Output
Output the title of this problem, "This Is Correct!". Include the exclamation point, but don't include the quotation marks.

Example Output To Screen
```
This Is Correct!
```

Problem 2 | **6 Points**

Test Scores

Program Name: scores.java **Input File: scores.dat**

The input file contains a table with student names in the second column and the score they received on a recent test in the first column. This is a strange way to format the table, so please write a program to reverse the columns so that the student names are in the first column and the scores are in the second.

Input
The input consists of 5 lines, each containing a student's score (from 0 to 100), a single space, and a student's first name (up to 20 characters).

Output
Output the same 5 lines with the names and scores reversed.

Example Input File
```
99 James
87 Tim
0 Marc
99 Laura
100 Buddy
```
Example Output To Screen
```
James 99
Tim 87
Marc 0
Laura 99
Buddy 100
```

Problem 3 | **6 Points**

List the Primes

Program Name: primes.java **Input File: primes.dat**

Write a program that will list all the prime numbers in a given range. Remember, a prime number is one whose only integer divisors are itself and one.

Input

The input will consist of 1 to 20 data sets, one per line. Each data set will consist of two integers from 2 to 1000 separated by a single space. The first integer will always be less than or equal to the second because they indicate a range of integers that you will be searching for primes.

Output

List, in ascending order, the primes that occur in the indicated range (including the endpoints). If there are no primes in the indicated range, print the message, "No primes found!".

Example Input File

```
2 10
20 20
100 120
```

Example Output To Screen

```
2 3 5 7
No primes found!
101 103 107 109 113
```

Problem 4	6 Points

Arena

Program Name: arena.java **Input File: arena.dat**

You are a programmer on a team working on a new RPG called Arena. You have been assigned to write the battle engine of the game. Write a program that will take in the attacks and blocks of two fighters and determine the victor of the battle.

Each battle will consist of 5 rounds, and each fighter starts a battle with 5 health. During each round, each fighter performs an attack and a block at one of three heights: low, medium, or high. If a fighter attacks a height that is not blocked by the other fighter, the attack inflicts damage of 2 health points, otherwise no damage is incurred. After a given round, if either fighter is out of health, the battle is over. If only one fighter survives a given round, that fighter is the winner. If both fighters die in the same round or both end the match with equal health, the battle ends in a draw. Otherwise, the victor is the fighter with the most remaining health.

Input

The first line of input will be a single integer indicating the total number of battles (from 1 to 20). Each battle will consist of 4 rows, each containing 5 integers separated by single spaces. The first and second rows indicate the first fighter's attack and block heights respectively, and the third and fourth rows contain similar information for the second fighter. Attack/block heights are encoded as numbers: 1 = high, 2 = medium, 3=low.

Output

For each battle, print a statement declaring the outcome of the battle. If there was a victor, output, "<Winner> is the victor!" where <Winner> is the fighter who has won the battle (either "Fighter 1" or "Fighter 2"). If the battle is a draw, output, "This battle ended in a draw!"

Example Input File

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

Example Output To Screen

```
Fighter 2 is the victor!
Fighter 2 is the victor!
This battle ended in a draw!
```

Problem 5 | **6 Points**

Dice Golf

Program Name: golf.java **Input File: golf.dat**

Here's a fun little 2-player dice game you can play to pass the time: The object of the game is to advance your peg to the tenth hole or beyond. Both players start their peg in the first hole, and taking turns rolling the dice, either advance or move back their pegs, depending on their roll. Each of the possible roll's values are:

```
Roll     Value     Description
--------------------------------
2          +9      Hole in one
3          -2      Sand trap
4          +2      On the green
5          +2      Long drive
6          +1      On the fairway
7          -3      Water hazard
8          +1      On the fairway
9          -2      In the rough
10         -2      Slice
11         +1      Nice chip shot
12         +1      Nice chip shot
```

Notes:

- It is not possible to move back beyond the first hole; any roll that would take a player backwards past the first hole takes the player's peg to the first hole instead.
- Any roll that would take a player to the tenth hole or beyond is a win for that player and the game is considered over.
- Player 1 always rolls first.

Input

Input to this problem will consist of a (non-empty) series of up to 100 data sets. Each data set will be formatted according to the following description, and there will be **no blank lines** separating data sets.

Each data set represents a game and has 2 lines:

1. *Number of rolls* - A single integer, N, the number of rolls in the game. 1 <= N <= 20.
2. *Rolls* - A list of N integers delimited by a single space. Each roll will be 2 <= X <= 12. Since Player 1 always rolls first, the first integer represents Player 1's first roll, the next integer represents Player 2's first roll (if he has one), the next integer represents Player 1's second roll (if he has one), and so on. There may be more rolls in the input than are necessary to determine the winner.

Note that the number of data sets is not explicitly given.

Output

Output will be a single line with the phrase: "Player X wins!", where X is the number of the player who won. Every game will have a winner.

Example Input File

```
2
2 2
3
3 5 2
15
6 3 8 5 12 4 4 4 4 5 11 4 5 2 7
```

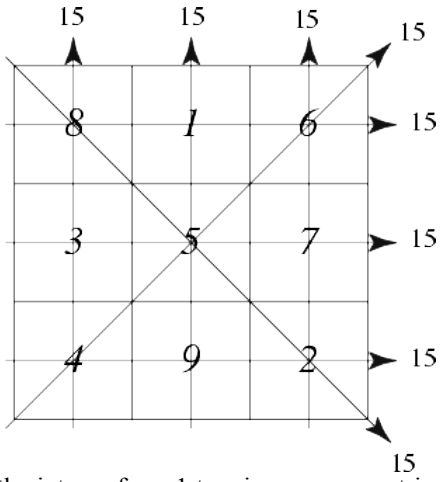
Example Output To Screen

```
Player 1 wins!
Player 1 wins!
Player 2 wins!
```

Problem 6 | **6 Points**

Matrix

Program Name: matrix.java **Input File: matrix.dat**



A magic square is formed by placing the integer from 1 to *n* in a square matrix where the sums of each row, column, and both diagonals are equal. Each integer from 1 to *n* must be used **exactly once** to form a magic square. In the illustration, each row, column, and diagonal sums to 15.

Write a program that will determine if a given matrix represents a magic square.

Input

The input will consist of up to 20 square matrices of dimension 1x1 to 10x10. The first line of the input file will contain an integer indicating the total number of matrices in the input. For each matrix, there will be a single line containing an integer, *n*, indicating the size of the matrix (*n* x *n*). The next *n* lines will contain *n* integers separated by spaces; this represents one of the matrices that is to be tested.

Output

For each matrix in the input, output a single line. If the matrix is a magic square, output, "This magic square has sum = <sum>." Replace <sum> with the number that is the sum of each row, column, and diagonal. If this matrix isn't a magic square, output, "This isn't a magic square." (Don't forget the periods!)

Example Input File

```
3
3
8 1 6
3 5 7
4 9 2
4
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
3
9 2 7
4 6 8
5 10 3
```

Example Output To Screen

```
This magic square has sum = 15.
This isn't a magic square.
This isn't a magic square.
```

Problem 7 | **6 Points**

Matrix Reloaded

Program Name: reloaded.java **Input File: reloaded.dat**

Write a program to determine if a matrix of integers with several "blanks" can have those blanks filled in to form a magic square. (For a description of magic squares, refer to the introduction of the problem titled *Matrix*.)

Input

The input will consist of up to 20 square matrices of dimension 1x1 to 10x10. The first line of the input file will contain an integer indicating the total number of matrices in the input. For each matrix, there will be a single line containing an integer, *n*, indicating the size of the matrix (*n* x *n*). The next *n* lines will contain *n* integers separated by spaces; this represents one of the matrices that is to be tested. Values of 0 in the matrix represent the blanks that need to be filled in. There will be no more than three values of 0 in any one input matrix.

Output

For each matrix in the input, output a single line. If the matrix can be made into a true magic square by filling in the blanks in any way, output, "This could be a magic square." Otherwise, output, "This can't be a magic square." (Don't forget the periods!)

Example Input File

```
2
3
8 1 6
3 0 7
4 9 2
2
0 2
3 0
```

Example Output To Screen

```
This could be a magic square.
This can't be a magic square.
```

Problem 8		6 Points

PacMan

Program Name: pacman.java Input File: pacman.dat

PacMan was popular many years ago, but now he's old and can't eat like he used to. These days he has rheumatism and just wants to find the quickest way out of each maze without getting eaten by a ghost. Luckily, the ghosts are older too (and lazy). They simply stake out key areas in each maze, hoping that PacMan will run into them accidentally. Of course, if PacMan eats a power pellet before hitting a ghost, it's the ghost that will be eaten. But the ghosts are lazy, and it's a risk they're willing to take.

Write a program that determines the least number of moves PacMan must make to reach the exit of each maze without being eaten. Each move is either one unit up, down, left, or right; PacMan doesn't have the dexterity to move diagonally.

Input

The input will consist of up to 20 square mazes of dimension 4x4 to 10x10. The first line of the input file will contain an integer indicating the total number of mazes in the input. For each maze, there will be a single line containing an integer, *n*, indicating the size of the maze (*n* x *n*). The next *n* lines represent the maze and will contain *n* characters each. Possible characters for each maze are:

- 'C' PacMan's starting position. Each maze will contain exactly one.
- 'X' Exit. Each maze will contain exactly one, and it is PacMan's goal to reach it.
- '#' Wall. This is impassible. The outside of each maze will always be surrounded by walls, but they can appear inside the maze as well.
- 'A' Ghost. Each maze will contain 0 to 4 ghosts. They do not move, and PacMan cannot pass them unless he eats (passes over) a power pellet first.
- '@' Power pellet. PacMan may pass through these, and if he does then he can pass through ghosts freely for the remainder of his moves in this maze. Each maze will contain 0 to 4 power pellets.
- '.' Empty space. This is a regular passable space. It has no special significance.

Output

If PacMan can get to the exit, print the message, "PacMan can escape in <X> moves." Replace <X> with the minimum number of moves required for PacMan to escape. If PacMan cannot get to the exit, print the message, "PacMan should retire."

Example Input File

```
2
5
#####
#C#X#
#.#.#
#.A.#
#####
7
#######
#.....#
#.###.#
#..X#.#
#####A#
#@...C#
#######
```

Example Output To Screen

```
PacMan should retire.
PacMan can escape in 20 moves.
```

Problem 9 **6 Points**

drawkcaBsay iPgay itaLnay Day

Program Name: day.java **Input File: day.dat**

The citizens of axeTsay have an unusual holiday they like to celebrate called drawkcaBsay iPgay itaLnay Day. On this day, the governor decrees that all citizens will speak in backwards pig latin. Naturally, this is a difficult task for many, and you are called in to write a translation program.

Input

Input to this problem will consist of a (non-empty) series of up to 100 data sets. A single data set is a single line containing a list of one or more words, delimited by a single space. Words are made only of alphabetic characters. There will be **no blank lines** separating data sets.

Output

For each data set, there will be exactly one line of output: the list of input words, each translated in backwards pig latin. To translate a word into backwards pig latin, do the following:

1. Reverse the order of the letters in the word.
2. Move the first letter of the new word to the end of the word.
3. Add the letters 'a' and 'y' to the end of the word.

Example Input File

```
Hello nice to meet you
This is a funny way to talk
```

Example Output To Screen

```
lleHoay cineay toay eemtay oyuay
ihTsay isay aay nnufyay awyay toay latkay
```

Problem 10 | **6 Points**

Roman Numeral Translator

Program Name: roman.java **Input File: roman.dat**

You are to write a program that reads in Roman numerals and determines its integer equivalent.

Roman numerals are based on letters where letters have the following values:

M = 1000
D = 500
C = 100
L = 50
X = 10
V = 5
I = 1

To translate a given number from Roman numerals to its decimal equivalent, the values of the letters are added from left to right. If a letter of lower value is immediately to the left of a letter of higher value, they are treated as a single unit with value equal to the value of the higher letter minus the value of the lower letter. No letter will ever have a lower letter immediately to the left and a higher letter immediately to the right or it would be ambiguous which letters were subtracted. For instance, IVX is not valid. There are other rules governing the use of Roman numerals, but they do not have to be known to perform the translation to decimal.

Input

The input will consist of 1 to 20 data sets. Each data set will consist of a string of letters that correspond to a Roman numeral. The max Roman numeral value will be MMM (3000), and no input value will contain more than 20 letters.

Output

For each data set, an integer corresponding to the Roman numeral will be printed, each on a separate line.

Example Input File

```
III
XIV
CCLXXXIX
```

Example Output To Screen

```
3
14
289
```