2020

Problem Packet

DO NOT OPEN THIS PACKET
UNTIL THE CONTEST BEGINS

This page is intentionally blank. Once the contest starts, you can remove this page.

2020 Problem Packet

Frequently Asked Questions

How does the contest work?

To solve each problem, your team will need to write a computer program that reads input from the standard input channel and prints the expected output to the console. Each problem describes the format of the input and the expected format for the output. When you have finished your program, you will submit the source code for your program to the contest website.

How is each problem scored?

Each problem is assigned a point value based on the difficulty of the problem. If the outputs match exactly, you will be given the points for the problem. There is no partial credit; your outputs must match *exactly*.

How are ties broken?

At the end of the contest, teams will be ranked based on the number of points they earned from correct answers during the contest. If there is a tie for the top three positions in either division, ties will be broken as follows:

1. Fewest problems solved (this indicates more difficult problems were solved)
2. Fewest incorrect answers (this indicates they had fewer mistakes)
3. First team to submit their last correct response (this indicates they worked faster)

Rounding

Some problems will ask you to round numbers. All problems use the “half up” method of rounding unless otherwise stated in the problem description. Most likely, this is the sort of rounding you learned in school, but some programming languages use different rounding methods by default. **Unless you are certain you know how your programming language handles rounding, we recommend writing your own code for rounding numbers based on the information provided in this section.**

With “half up” rounding, numbers are rounded to the nearest integer. For example:

4. 1.49 rounds down to 1
5. 1.51 rounds up to 2

The “half up” term means that when a number is exactly in the middle, it rounds to the number with the greatest absolute value (the one farthest from 0). For example:

6. 1.5 rounds up to 2
7. -1.5 rounds down to -2

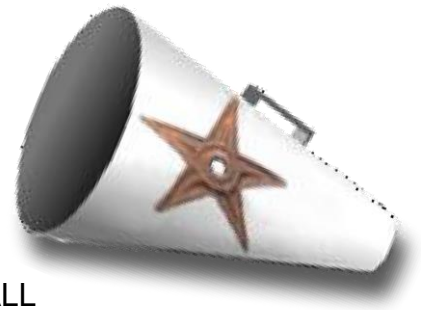
Rounding errors are a common mistake; if a problem requires rounding and the contest website keeps saying your program is incorrect, double check the rounding!

Terminology

Throughout this packet, we will describe the inputs and outputs your programs will receive. To avoid confusion, certain terms will be used to define various properties of these inputs and outputs. These terms are defined below.

8. An **integer** is any whole number; that is, a number with no decimal or fractional component: -5, 0, 5, and 123456789 are all integers.
9. A **decimal number** is any number that is not an integer. These numbers will contain a decimal point and at least one digit after the decimal point. -1.52, 0.0, and 3.14159 are all decimal numbers.
10. **Decimal places** refer to the number of digits in a decimal number following the decimal point. Unless otherwise specified in a problem description, decimal numbers may contain any number of decimal places greater or equal to 1.
11. A **hexadecimal number** or **string** consists of a series of one or more characters including the digits 0-9 and/or the uppercase letters A, B, C, D, E, and/or F. Lowercase letters are not used for hexadecimal values in this contest.
12. **Positive numbers** are those numbers strictly greater than 0. 1 is the smallest positive integer; 0.000000000001 is a very small positive decimal number.
13. **Non-positive numbers** are all numbers that are not positive; that is, all numbers less than or equal to 0.
14. **Negative numbers** are those numbers strictly less than 0. -1 is the greatest negative integer; -0.000000000001 is a very large positive decimal number.
15. **Non-negative numbers** are all numbers that are not negative; that is, all numbers greater than or equal to 0.
16. **Inclusive** indicates that the range defined by the given values includes both of the values given. For example, the range 1 to 3 inclusive contains the numbers 1, 2, and 3.
17. **Exclusive** indicates that the range defined by the given values does not include either of the values given. For example, the range 0 to 4 exclusive includes the numbers 1, 2, and 3; 0 and 4 are not included.
18. **Date and time formats** are expressed using letters in place of numbers:
    - **HH** indicates the hours, written with two digits (including a leading zero when needed). The problem description will specify if 12- or 24-hour formats should be used.
    - **MM** indicates the minutes for times or the month for dates. In both cases, the number is written with two digits (including a leading zero when needed). January is month 01.
    - **YY** or **YYYY** is the year, written with two or four digits (including a leading zero when needed).
    - **DD** is the date of the month, written with two digits (including a leading zero when needed).

Problem 1: No More Shouting



Points: 5

Problem Background

It's common knowledge that on the internet, TYPING IN ALL UPPERCASE LETTERS ISN'T VERY POLITE. It just looks like you're shouting at people, which isn't a very good way to hold a conversation. You've been asked to design a browser extension that can (forcibly) calm everyone down by converting UPPERCASE SHOUTING into lowercase whispers. Try to stay calm as you solve this problem.

Problem Description

Your program will be given lines of text in which all letters are uppercase. You must convert these letters to lowercase without otherwise changing the content of the text.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include a single line of text consisting of uppercase letters, numbers, spaces, and/or punctuation.

```
2
THIS SENTENCE IS IN ALL CAPS
SHOUTING ISN'T NICE.
```

Sample Output

For each test case, your program must output the provided string after replacing all uppercase letters with their lowercase equivalents. Spaces, numbers, and punctuation should not be modified.

```
this sentence is in all caps
shouting isn't nice.
```

Problem 2: Sum It Up

Points: 5

Problem Background

Adding up numbers is very easy, unless you add a twist. If two numbers are the same, sum their sums!

Problem Description

Your program will be given two numbers. If they are not equal, return their sum. If they are equal, return double their sum.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include a single line consisting of two non-negative integers separated by spaces.

```
5
1 3
2 2
3 2
13 13
125 9
```

Sample Output

For each test case, your program must output the value calculated according to the rules described above.

```
4
8
5
52
134
```

Problem 3: Goofy Gorillas

Points: 5

Problem Background

The local zoo's most popular exhibit contains two gorillas. However, the gorillas can cause the zookeepers some issues. We need to be able to alert the zookeepers of trouble in the gorilla compound.

Problem Description

Your program will be given information about whether each of the gorillas is smiling or not. We need to alert the zookeepers if both gorillas are smiling (which might mean they're causing trouble), or if neither gorilla is smiling (which might mean they're about to fight). If only one gorilla is smiling, everything is probably ok.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include a single line containing two boolean values ("true" or "false") separated by spaces.

```
2
true false
true true
```

Sample Output

For each test case, your program must output "true" if the zookeepers should be alerted about potential trouble, or "false" if everything seems ok.

```
false
true
```

Problem 4: Brick House

Points: 10

Problem Background

We want to build a row of bricks for our brick house that is a certain number of inches long, and we have a number of small bricks and large bricks with which to do it. You need to write an application that will decide if its is possible to build this row of bricks using some or all of the given bricks. You do not need to use all of the given bricks!

Problem Description

Your program will be given a goal length for the brick wall and the number of small and large bricks available. Small bricks are each 1 inch long. Large bricks are 5 inches long. You will need to determine if it is possible to build a row of bricks exactly as long as the goal using only the available bricks.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will consist of a single line, including three non-negative integers separated by spaces:

- The first integer represents the number of small, one-inch-long bricks
- The second integer represents the number of large, five-inch-long bricks
- The third integer represents the target length of the wall, **X**, in inches

```
3
3 1 8
3 1 9
3 2 10
```

Sample Output

For each test case, your program must print a single line with the word "true" if it is possible to build a wall of exactly **X** inches using only the bricks available. Otherwise, it should print "false".

```
true
false
true
```

Problem 5: Image Compression

Points: 15

Problem Background

Images can be saved onto a computer in many different types of file formats, each with its own advantages and disadvantages. JPEG (or JPG) images are commonly used for photography, because their format allows the image information to be compressed, reducing the size of the file and allowing you to take more pictures. The downside to this is that repeatedly editing a JPEG image causes the quality of the image to gradually get worse over time; each time the file is saved, the existing image data is compressed further and further, losing fine details.

The process of compressing a JPEG image is complicated but can be broken down into several individual steps. One of these steps is called quantization, which takes a wide range of numbers created by a previous step in the process and converts them to a smaller, more manageable scale. This results in some loss of detail as previously mentioned; two different but close numbers may be converted to the same result number. However, the human eye often cannot discern very high-frequency changes, so this loss is usually not noticeable.

Problem Description

Your program will need to implement an example quantization algorithm that accepts perceived brightness values and converts them to an integer value between 0 and 255 inclusive. Your program will be given a list of decimal values representing brightness values (such as might be read by a scanner). Your program must identify the highest (max) value and the lowest (min) value from the list of values, then convert all values in the list to the target scale using this formula:

$$Output = \frac{Input - Min}{Max - Min} * 255$$

All results should be rounded to the nearest integer.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include the following lines of input:

- A positive integer, **X**, representing the number of values in the list
- **X** lines, each containing a decimal number to be converted

```
2
5
0.0
25.0
50.0
75.0
100.0
6
12.3
-67.1
122.8
428.4
-15.9
221.0
```

Sample Output

For each test case, your program must output the list of converted numbers, maintaining the same order. Print one number per line, and round all results to the nearest integer.

```
0
64
128
191
255
41
0
98
255
26
148
```

Problem 6: Foveated Rendering

Points: 15

Problem Background

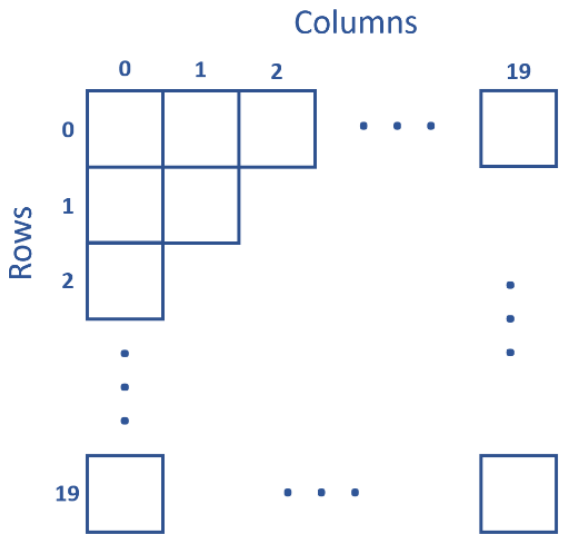
Virtual Reality has exploded into the market in the last five years, being used for everything from games and entertainment to product design and engineering. One of the more recent advances in VR headset design is the addition of eye tracking to increase performance.

The human eye has an extremely narrow field of view in which perfect 20/20 vision is attainable and fine detail can be distinguished. This clarity of vision is due to the fovea, a small depression in the inner retina specialized for this purpose. However, due to the size of the fovea, the human eye can only see clearly within a field of view of less than 10°. The rest of our vision comes from the brain piecing together imagery as we look around.

Due to this fact, a VR headset only needs to render the highest resolution imagery directly where the user is looking. Images outside of that field of view can be rendered at a lower quality, increasing the performance of the system.

Problem Description

You have been tasked with writing a module for a virtual reality application that determines the rendering quality for each section of the headset's screen. For simplicity, your module will only deal with a single eye on a single screen. The screen will be divided into a 20-by-20 grid of blocks.



Your program will be given the coordinates within the grid at which the user is currently focusing their sight, and will need to output the rendering level of each cell in the grid row by row.

The cell the user is looking directly at should be rendered at full quality - 100%. All cells around that cell should be rendered at half quality (50%), and all cells around those should be rendered at one-quarter quality (25%). All other cells should be rendered at the minimum level of 10%.

For example, if the user is looking at the block in row 7, column 10, the rendering quality for each block in the grid would be:

Row \ Col	0	⋯	7	8	9	10	11	12	13	⋯	19
0	10%	⋯	10%	10%	10%	10%	10%	10%	10%	⋯	10%
⋮	⋮	⋱	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋰	⋮
4	10%	⋯	10%	10%	10%	10%	10%	10%	10%	⋯	10%
5	10%	⋯	10%	25%	25%	25%	25%	25%	10%	⋯	10%
6	10%	⋯	10%	25%	50%	50%	50%	25%	10%	⋯	10%
7	10%	⋯	10%	25%	50%	100%	50%	25%	10%	⋯	10%
8	10%	⋯	10%	25%	50%	50%	50%	25%	10%	⋯	10%
9	10%	⋯	10%	25%	25%	25%	25%	25%	10%	⋯	10%
10	10%	⋯	10%	10%	10%	10%	10%	10%	10%	⋯	10%
⋮	⋮	⋰	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮
19	10%	⋯	10%	10%	10%	10%	10%	10%	10%	⋯	10%

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include a single line of input containing two integers, separated by spaces, representing the row and column number of the eye position within the screen's grid, respectively. Row and column numbers will be between 0 and 19 inclusive.

```
2
7 10
0 0
```

Sample Output

For each test case, your program must output the rendering quality percentage for each block in the grid. Each row should be printed as a separate line, and columns should be separated by spaces.

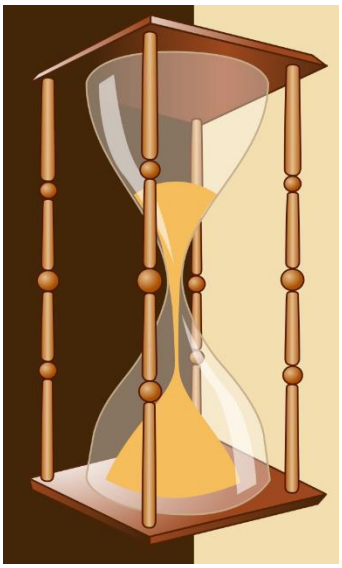
```
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 25 25 25 25 25 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 25 50 50 50 25 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 25 50 100 50 25 10 10 10 10 10 10 1(
10 10 10 10 10 10 10 10 25 50 50 50 25 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 25 25 25 25 25 10 10 10 10 10 10 10
```

10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
100	50	25	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
50	50	25	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
25	25	25	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10

Problem 7: Time and Time Again

Points: 20

Problem Background



Times and periods of times can be expressed in many different ways. National and regional differences, and even personal preferences, have led to a wide range of formats for expressing times. This can lead to a great deal of confusion; does the date 01/03 refer to January 3rd or March 1st… or January 2003? Is the time 8:45 in the morning or the evening?

You have been asked to break through some of this confusion by converting a list of times to a new, consistent format.

Problem Description

Your program will receive a list of time durations that provide the number of hours, minutes, and/or seconds within the duration.

- Hours will be given as a non-negative integer followed by a lowercase letter ‘h’ (e.g. 2h). Hours will range from 0 to 99 inclusive.
- Minutes will be given as a non-negative integer followed by a lowercase letter ‘m’ (e.g. 2m). Minutes will range from 0 to 59 inclusive.
- Seconds will be given as a non-negative integer followed by a lowercase letter ‘s’ (e.g. 2s). Seconds will range from 0 to 59 inclusive.

These values may not be presented in this order. Values may be separated by spaces, commas, and/or the word “and”; this text should be ignored. Some of these values may be missing; for example, an input may only give you minutes and seconds. Any omitted values should be assumed to be zero.

Regardless of what information is provided, your program will need to print the duration in a simpler, more consistent format:

HH:MM:SS

In this format, HH is a two-digit number representing the number of hours (including a leading zero, if necessary). MM is a two-digit number representing the number of minutes (including a leading zero, if necessary). SS is a two-digit number representing the number of seconds (including a leading zero, if necessary). Each number is

separated from the next with a colon, and they are always presented in the same order. All numbers must be included with the output, even if they are zero.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include a single line of input containing a string describing a time duration in a variable format as noted above.

```
5
1m and 45s
10m,10s
32s, and 12h
76h
1s
```

Sample Output

For each test case, your program must output the same time interval on a single line in the HH:MM:SS format described above.

```
00:01:45
00:10:10
12:00:32
76:00:00
00:00:01
```

Problem 8: Caesar Cipher

Points: 20

Problem Background

The Caesar Cipher is one of the earliest known ciphers, and among the simplest to learn. It is a “substitution cipher”, in which each letter in the original message (the “plaintext”) is shifted a certain number of places down the alphabet. For example, with a shift of 1, an A would be replaced with a B, a B would be replaced with a C, and so on. This method is named after Julius Caesar, who apparently used it to communicate with his generals.

To pass an encrypted message from one person to another, it is necessary that both parties have the “key” for the cipher, so that the sender can encrypt it and the recipient can decrypt it. For the Caesar Cipher, the key is the number of letters by which to shift the cipher alphabet.

Problem Description

You are working for the History Channel, who wants to decrypt all communications that Julius Caesar made to his generals in order to support a new documentary they’re filming about the Roman emperor. You will be given a list of encrypted messages, and the key believed to be used to encrypt those messages. Your program must decrypt those messages.

For the purposes of this problem, we will be using the English alphabet, shown below in its standard order (with a shift of 0).

```
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
```

If encrypting a message with a shift of 1, each letter in the plaintext will be replaced with the respective letter shown in the 1-shifted alphabet below.

```
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 A
```

To decrypt a message, the process is reversed; a letter in the ciphertext would be replaced with the respective letter in the original English alphabet.

Spaces are not encrypted in this cipher and should remain in place when decrypting a message.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include two lines:

- A line with a single integer representing the message key - the number of letters by which to shift the alphabet when encrypting the message.
- A line containing lowercase letters and spaces, representing the encrypted message.

```
3
1
buubdl bu ebxo
3
ghvwurb wkh fdvwoh
6
yzkgr znk ynov
```

Sample Output

For each test case, your program must output the decrypted message. Messages should be printed in lowercase, and all spaces should be retained.

```
attack at dawn
destroy the castle
steal the ship
```

Problem 9: Count to 10

Points: 25

Problem Background

When testing software or hardware, it's considered a "best practice" to test every possible situation to prove that the code or device is stable under any condition it might come across. For example, if we have a chip with eight LEDs, we might want to light up those LEDs in every combination to make sure they function properly. This is essentially an 8-bit binary counter, displaying each number from 0 to 255.

Problem Description

In this problem, you will need to generate test data for a binary counter like that described above. You will be provided with the number of bits to use for your counter, and will need to generate a list of all binary numbers with at most that number of bits in numerical order.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include a single line with a positive integer, representing the number of bits to use.

```
1
3
```

Sample Output

For each test case, your program must output a list of binary numbers, ranging from 0 to the maximum value with the indicated number of bits, inclusive. Numbers must be listed one per line, in numerical order. Include any leading zeros up to the required bit length.

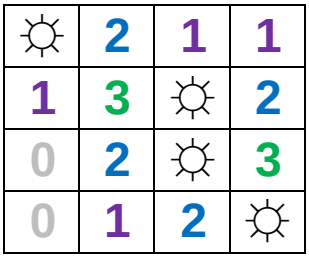
```
000
001
010
011
100
101
110
111
```

Problem 10: Minesweeper

Points: 25

Problem Background

Minesweeper is a type of single-player puzzle game in which the player continuously selects different cells of a rectangular grid. Each cell of the grid is either occupied by a bomb or is a safe cell. If the player selects a cell occupied by a bomb, they "explode" and lose the game. Otherwise, the selected cell shows the number of neighboring cells that contain bombs. Cells are neighbors if they are adjacent horizontally, vertically, or diagonally.



Problem Description

You will need to write a program that receives the size of a minesweeper grid and the locations of the mines within that grid, then uses that information to display the completed grid. The output should include the locations of all bombs, and numbers in the safe cells indicating the number of neighboring bombs.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include:

- A line containing three positive integers separated by spaces, representing:
  - The number of rows within the minesweeper grid, **R**
  - The number of columns within the minesweeper grid, **C**
  - The number of bombs within the minesweeper grid, **B**
- **B** lines representing the location of each bomb within the grid. Each line contains two integers separated by spaces, representing:
  - The row of the bomb's cell. The topmost row in the grid is row 0. Values will range from 0 (inclusive) to **R** (exclusive).
  - The column of the bomb's cell. The leftmost column in the grid is column 0. Values will range from 0 (inclusive) to **C** (exclusive).

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

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

Sample Output

For each test case, your program must output the minesweeper grid described by the input. Write each row on a separate line, and one character per cell. Cells containing bombs should be represented by an asterisk character (*); safe cells should contain a number (0 through 8 inclusive) equal to the number of bombs in neighboring cells.

```
*2
2*
011
02*
02*
232
**1
```

Problem 11: Homeward Bound

Points: 30

Problem Background

After a long delay figuring out what route he should take, the travelling salesman has just finished his journey and has collected orders from customers all along the way. He finished his trip at his company's warehouse, and now he wants to deliver all of his customer's orders on the way back home. To make sure he doesn't miss anyone, he's planning to take the same trip in reverse order. Unfortunately, as he's walking towards the ticket counter at the airline, he trips, and scatters his used boarding passes everywhere! He needs your help to get the boarding passes back in the correct order so he can reconstruct his journey and figure out how to get back home.

Problem Description

Your program will be given several pairs of cities, indicating a departure and arrival point for each of the salesman's boarding passes. These pairs will be out of order. You must reconstruct his original journey by determining the correct order of boarding passes, then print the route he should take back home (the original route in reverse order).

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include the following lines of input:

- A positive integer, **X**, indicating the number of boarding passes
- **X** lines, each listing two city names, separated by spaces. The first city is the departure city for that boarding pass; the second is the arrival city. City names will consist of upper and lower-case letters and underscores (_).

```
2
4
Fort_Worth Denver
Washington Toronto
Orlando Fort_Worth
Denver Washington
5
Riyadh Singapore
Madrid London
Chicago Madrid
Berlin Riyadh
London Berlin
```

Sample Output

For each test case, your program must output each city the salesman should visit on the way home, one city per line, starting with his original final destination.

```
Toronto
Washington
Denver
Fort_Worth
Orlando
Singapore
Riyadh
Berlin
London
Madrid
Chicago
```

Problem 12: Conway's Game of Life

Points: 40

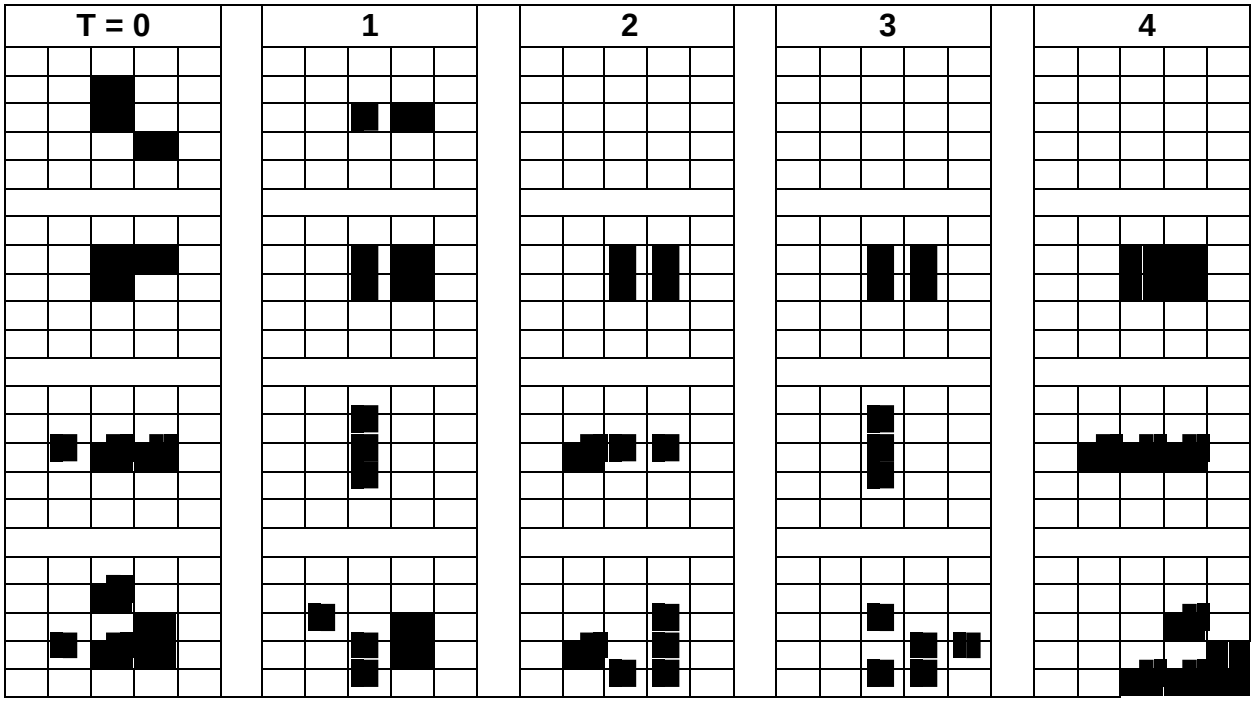
Problem Background

In 1940, computer scientist John von Neumann defined life as a creation which can reproduce itself and simulate a Turing machine: briefly, a device which acts according to a set of rules. This definition gave rise to a continuing series of mathematical experiments. Among the most famous of these is a "game" created by mathematician John Conway in 1970 called *Life*. Conway's *Life* consists of a set of four rules to be followed by a computer given an initial state of a grid filled with "live" and "dead" cells.

In each generation:

1. Any live cell adjacent to one or zero live cells dies (from loneliness).
2. Any live cell adjacent to two or three live cells lives.
3. Any live cell adjacent to four or more live cells dies (from overcrowding).
4. Any dead cell adjacent to exactly three live cells becomes alive (through reproduction).

Diagonal cells are considered to be adjacent. *Life* evolves by applying these rules to the "world" represented by the grid. The rules are applied, the world is redrawn, the rules are applied again, and the world is redrawn again, repeating indefinitely



These seemingly simple rules are completely deterministic; that is, each generation is determined entirely by the state of the previous generation. Despite this, these rules can yield some very complex behavior. Theoretically, *Life* is a "universal Turing machine;" this means that anything that can be calculated through an algorithm can be calculated with *Life*.

Problem Description

You must design a program that implements Conway's *Life* on a 10-by-10 grid. Your program will be given an initial state for the first generation. It must then determine the state of the world after a given number of generations have been performed. Note that cells outside the bounds of the 10-by-10 grid are always considered dead.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include the following lines of input:

- A line containing a positive integer, **X**, indicating the number of generations to calculate
- Ten lines containing ten characters each representing the initial state of the world. Characters will be either '1', representing a "live" cell, or '0', representing a "dead" cell.

```
1
6
0000000000
0000000000
0000000000
0000010000
0000111000
0000111000
0000010000
0000000000
0000000000
0000000000
```

Sample Output

For each test case, your program must output the state of the world after the indicated number of generations. Each test case should include ten lines with ten characters each.

```
0000000000
0000000000
0000111000
0001000100
0000000000
0000000000
0001000100
0000111000
0000000000
0000000000
```

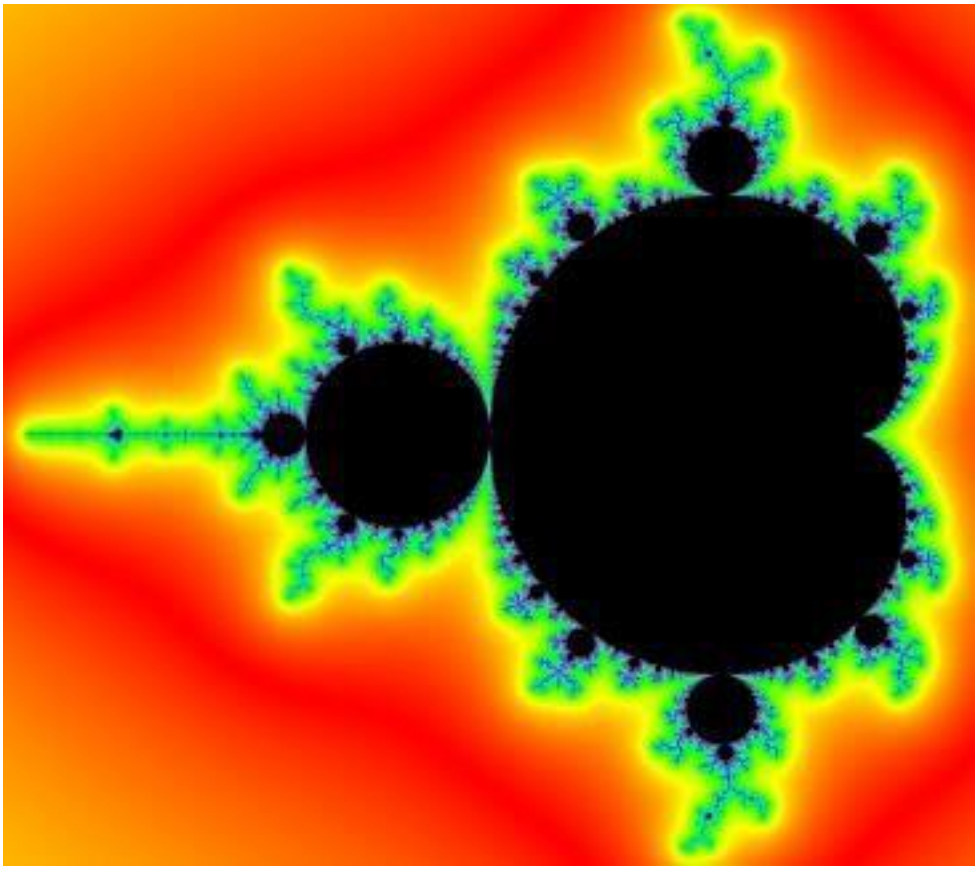
Problem 13: Mandelbrot Set

Points: 45

Problem Background

The Mandelbrot set is drawn by considering the recursive function $Z_{n+1} = {Z_n}^2 + c$, where c is a complex number of the form $a + bi$ (in mathematics, i is an imaginary number with the value of $\sqrt{-1}$; thus, $i^2 = -1$). By iterating repeatedly, using each value of Z to calculate the next value, we find that for some input values of c, Z grows without bound. For others, Z remains bound.

To draw the Mandelbrot set, we use the "complex plane", where the horizontal x-axis represents the value of a, and the vertical y-axis represents the value of b. Each point is colored based on the number of iterations (n) we can perform before the absolute value of Z ($|Z_n|$) becomes greater than a specified value. When this happens, it is said that the function "diverges". In the image below, black indicates that $|Z_n|$ remained below a prescribed value for all values of n. Blue pixels represent points at which it took many iterations to get $|Z_n|$ above that value; red pixels required fewer iterations.



Let's consider the function using a value of $c = 1.1 + 2i$.

Regardless of the value of c, the value of Z_0 always equals 0. We can use this to determine the value of Z_1:

$$Z_1 = Z_0^2 + c$$
$$Z_1 = 0^2 + 1.1 + 2i$$
$$Z_1 = 1.1 + 2i$$

From this, we can see that for any value of c, $Z_1 = c$. Now we need to determine if the function has diverged. For the purposes of this problem, we'll consider the function to have diverged if $|Z_n| \geq 100$. Since i is an imaginary number, we use this formula to determine the absolute value of numbers of the form $a + bi$:

$$|Z_1| = \sqrt{a_1^2 + b_1^2}$$
$$|Z_1| = \sqrt{1.1^2 + 2^2}$$
$$|Z_1| = \sqrt{1.21 + 4}$$
$$|Z_1| \approx 2.2825$$

2.2825 is less than 100, so the function hasn't diverged yet. We need to do more iterations to determine when it diverges, if ever:

$$Z_2 = Z_1^2 + c$$
$$Z_2 = (a_1 + b_1 i)^2 + a_0 + b_0 i$$
$$Z_2 = (1.1 + 2i)^2 + 1.1 + 2i$$
$$Z_2 = 1.1^2 + 1.1(2i) + 1.1(2i) + (2i)^2 + 1.1 + 2i$$
$$Z_2 = 1.21 + 4.4i - 4 + 1.1 + 2i$$
$$Z_2 = -1.69 + 6.4i$$
$$a_2 = -1.69$$
$$b_2 = 6.4$$
$$|Z_2| = \sqrt{-1.69^2 + 6.4^2}$$
$$|Z_2| \approx \sqrt{2.8561 + 40.96}$$
$$|Z_2| \approx 6.6194$$

(Remember that $i^2 = -1$, so above, $(2i)^2 = 2^2 * i^2 = 4 * -1 = -4$.)

$|Z_2|$ is still less than 100, so it hasn't diverged yet. How many iterations do we need to do to reach that point?

n	Z	a	b	$\|Z\|$
1	1.1 + 2*i*	1.1	2	2.2825
2	-1.69 + 6.4*i*	-1.69	6.4	6.6194
3	-37.0039 - 19.632*i*	-37.0039	-19.632	41.8892
4	984.9732 + 1454.9211*i*	984.9732	1454.9211	1756.9769

So at $n = 4$, we see that the value of $|Z| > 100$. This means that for this value of c, the function has diverged at 4. We color the point at x = 1.1, y = 2 an appropriate color for that value, and move on to the next value of c to be checked.

Problem Description

Your program must identify the color to use in a rendering of the Mandelbrot set for a given value of c. Use the following table and the explanation above to determine what colors should be used:

Value of n when function diverges	Color
≤ 10	RED
11-20	ORANGE
21-30	YELLOW
31-40	GREEN
41-50	BLUE
≥ 51	BLACK

For the example calculation above, the function diverged at $n = 4$, so the color for that value of c should be red.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include a single line of input with two decimal numbers separated by spaces. These numbers represent the values for a and b, respectively. Remember that $c = a + bi$.

```
4
1.1 2.0
-0.7 0.2
-0.5 0.65
-0.5 0.608
```

Sample Output

For each test case, your program must output the value of c, followed by a space, followed by the color used to render that value of c according to the table above. The color should be printed in uppercase letters. Decimal values should be printed as they were received from the input.

```
1.1+2.0i RED
-0.7+0.2i BLACK
-0.5+0.65i ORANGE
-0.5+0.608i BLUE
```

Problem 14: Network Ranger

Points: 50

Problem Background

How is the internet like the post office? They both use addresses!

Computers and other devices that connect to the internet are assigned an Internet Protocol (IP) address when they connect. While a newer format is available, most systems still use the IPv4 format for these addresses. In this format, an IP address consists of four numbers, separated by periods. Each number can range from 0 to 255. For example, the address 127.0.0.1 always represents your own computer (the "localhost").

As with any other piece of data, your computer stores these addresses in a binary format. Each number in the address is represented by an eight-bit binary string of 0's and 1's; these strings are concatenated with each other to form the full address. For example, the IP address 166.23.250.209 is converted as:

166								23								250								209							
1	0	1	0	0	1	1	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	0	1	0	1	1	0	1	0	0	1

Just like mailing addresses can be grouped by a ZIP code or postal code, IP addresses can be grouped by blocks. Internet companies can reserve these blocks to use in assigning IP addresses to their customers, through a system called Classless Inter-Domain Routing (CIDR). A CIDR block is defined by writing an IP address followed by a slash and the number of bits that match between all members of the block (on the left side of each address). For example, the IP addresses 192.168.0.0 and 192.168.108.68 are represented as the following binary numbers:

192								168								0								0							
1	1	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
192								168								108								65							
1	1	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	1	0	1	1	0	0	0	1	0	0	0	0	0	1

The first 17 bits of both addresses are the same, so these addresses are part of the 192.168.0.0/17 block (any further matches after the first mismatch aren't counted). This could also be written as the 192.168.108.65/17 block, but the convention is to use the first (smallest) address in a block when writing it out in this manner.

Blocks can be any size from /0 to /32. A /32 block would require that all 32 bits match; this represents a single address. A /0 block wouldn't require that any bits match; this represents the entire internet!

For this problem, you are working with the FBI's cyber crimes division to track down a ring of internet scammers using ransomware to attack innocent people. You've been able to track down a list of IP addresses used by the scammers. The FBI wants to get a search warrant to figure out who is behind these IP addresses, but a judge won't issue the warrant unless you can identify the smallest possible range that covers all of those addresses.

Problem Description

Your program will be given a list of IPv4 addresses and must identify the smallest CIDR block that contains every address. Each CIDR block should be written using the first (smallest) address within the block; that is, 192.168.0.0/16 may be an acceptable answer, but 192.168.0.1/16 is not.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include the following lines of input:

- A line containing a single positive integer, **X**, indicating the number of IP addresses used in this test case
- **X** lines, each containing a single IPv4 address

```
2
2
192.168.0.0
192.168.255.255
4
32.73.94.16
32.73.89.172
32.73.95.210
32.73.92.82
```

Sample Output

For each test case, your program must output the smallest CIDR range that contains every listed IP address, using the format described above.

```
192.168.0.0/16
32.73.88.0/21
```

Problem 15: Hide Your Spies

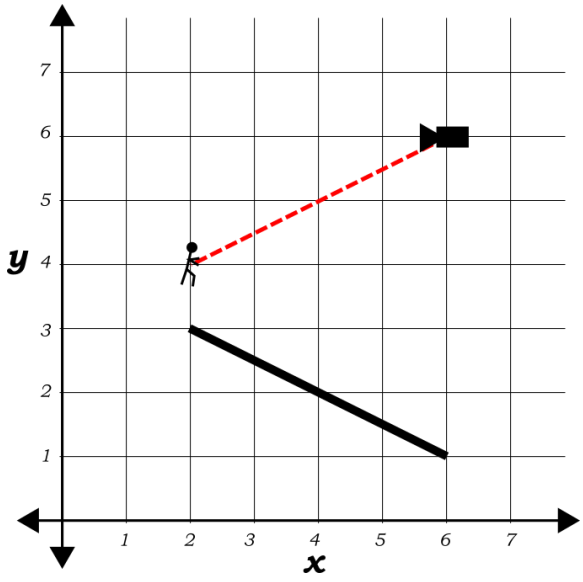
Points: 55

Problem Background

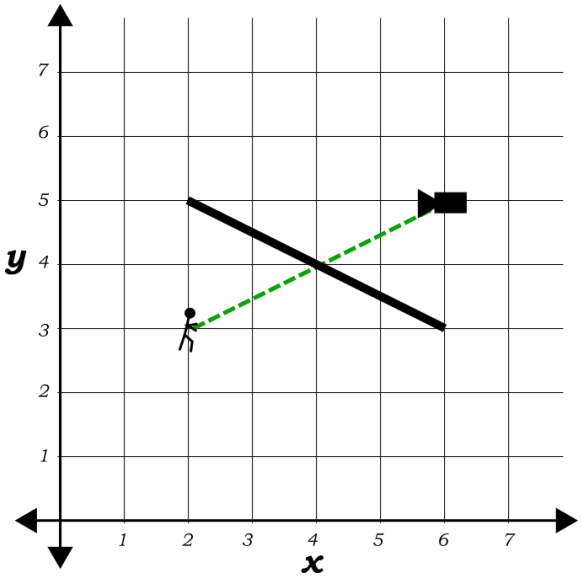
You're working with an intelligence agency to guide a spy through a secret enemy installation. The enemy has cameras positioned throughout the building with a 360° field of view; if your spy is caught on camera, the mission will fail! Fortunately, there are a number of walls blocking the view of the cameras that your spy can hide behind. You need to be able to determine if your spies will be seen based on the position of the cameras, spies, and the walls in the room.

Problem Description

Your mission, should you choose to accept it, is to determine if there is a clear line of sight from a camera at a given set of (x, y) coordinates to a spy located at a different set of coordinates. Several walls will be positioned throughout the room; if a wall intersects a line drawn between the camera and the spy, the spy is hidden and avoids detection. You must write a program that checks if the spy is successfully hidden and reports if he has been detected or not.



The wall's line doesn't intersect the line between the spy and the camera. The spy is detected!



The wall is between the camera and the spy, intersecting that line. The spy remains hidden.

To determine if two lines intersect, you'll need to locate the point at which the lines would intersect if they were continued infinitely in both directions. Remember that a (non-vertical) line can be defined using the equation

$$y = ax + c$$

a is known as the "slope" of the line, and can be calculated from any two points (x_1, y_1) and (x_2, y_2) as follows:

$$a = \frac{y_2 - y_1}{x_2 - x_1}$$

(If $x_2 - x_1 = 0$, then a is undefined, and the line is vertical.) Once you know a, you can calculate c using it and the (x, y) coordinates of one of the points on the line:

$$c = y - ax$$

Complete this process for the two lines you're trying to check for intersection to obtain both of their line equations. You can then use both equations to calculate the (x, y) point at which the lines would intersect. If this point is within the bounds of the points you already knew about, then the wall is blocking the camera's line of sight!

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include the following lines of input:

- A line containing five integers separated by spaces, representing the following information, in order:
  - The X-coordinate of the spy within the current room
  - The Y-coordinate of the spy within the current room
  - The X-coordinate of the camera within the current room
  - The Y-coordinate of the camera within the current room
  - The number of walls in the current room, **W**
- **W** lines containing four integers separated by spaces, each line representing information about a wall within the room:
  - The X-coordinate of the start of the wall
  - The Y-coordinate of the start of the wall
  - The X-coordinate of the end of the wall
  - The Y-coordinate of the end of the wall

```
2
2 2 6 4 1
2 5 5 5
```

```
2 2 6 4 2
4 1 4 5
1 5 4 5
```

Sample Output

For each test case, your program must output a single line containing either the word “YES” (indicating that the spy was seen by the camera) or “NO” (if the spy evaded detection).

```
YES
NO
```

Problem 16: Evacuate!

Points: 70

Problem Background

It's your first day working at as a software engineer. You've finished your orientation and are at your new desk, ready to start work when…

BEEP! … BEEP! … BEEP!

It's the fire alarm! You're not familiar with the building yet and don't know where to go! Fortunately, your coworkers help you get outside safely, and it was just a fire drill anyway, but the experience gives you an idea. What if you had an app on your phone that could guide you to the nearest exit? You present the idea to your manager, and they agree to start the project!

Problem Description

Your program will read in an image of a building's floor plan and must find the shortest route to the outside of the building from the given start position. While searching for the shortest path, you may travel in any cardinal direction - up, down, left, or right. You may not move diagonally, nor through walls. In the event that multiple paths are tied for the shortest length, take the path that exits closest to the top-left corner of the map. While the map will be rectangular (or square), the building's layout may not be.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include the following lines of input:

- A line containing two integers separated by spaces:
  - The first integer represents the width of the map, **W**
  - The second integer represents the height of the map, **H**
- A total of **H** lines, each up to **W** characters long, containing the map of the building.
  - A # (hashtag) character represents a wall of the building.

- A space indicates an empty navigable hallway or room.
- A lowercase letter o represents your start position within the building.
- An uppercase letter X represents an exit from the building.
- Lines may be shorter than **W** characters; any "missing" characters will be outside the building and should have no bearing on your work. Remember not to print any trailing whitespace in your output.

```
2
10 10
########X#
X     ## #
#####    #
#   # ## #
# # # ## #
#o# # # #
###  #  ##
#### ##  X
X       ##
######X###
30 20
##X###############
##            ####
##### ## ####    #
#   # ## #   ### #
# #  ## # # ### #
# ### #    #    #
#     # ## #######
# ##### ##     ###
#    ######      #
##### ####     # #
#     #### ##### #
##  ####### #    #############
##       # ######           #
#    # ## #      ########## #
### # ## # ##### ##         #
# ### #  # #    #  ##########
#     # ## #o#####          #
# ######## ################# #
#               #           #
##################X###########
```

Sample Output

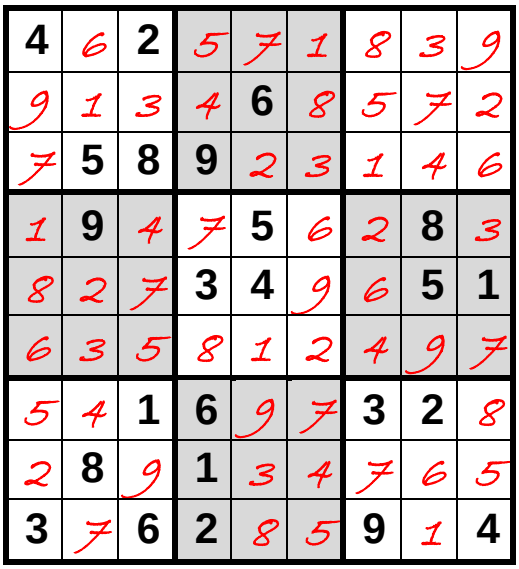
For each test case, your program must output the original map of the building, with the shortest path marked using periods (.) in place of the spaces presented above.

```
########X#
X     ## #
#####    #
#...# ## #
#.#.# ## #
#o#.#  #  #
###.. # ##
####.##  X
X  ...  ##
######X###
##X###############
##.......     ####
##### ##.####    #
#  # ##.#   ### #
# #  ##.# # ### #
# ### # ...#    #
#    # ##.#######
# ##### ##.    ###
#       ######.  #
##### ####.....# #
#     ####.##### #
##  #######.#    ##############
##       #.######            #
# #  ##  #.......  ########## #
### # ## # #####.##          #
# ### # # #.....#  ##########
#     # ## #o#####           #
# ######## ################# #
#               #           #
##################X###########
```

Problem 17: Sudoku

Points: 80

Problem Background



Sudoku is a popular logic puzzle commonly found in newspapers, magazines, and online. Most likely originating in Indiana in 1979, the puzzle format found great popularity in Japan in the 1980s and became a worldwide phenomenon in the new millennium. Newspapers in particular contributed to the puzzle's establishment as a household name due to the puzzle's similarities with crossword puzzles.

Sudoku is played on a 9-by-9 grid of squares divided into 3-by-3 subgrids. Each square is filled in with one of the numbers from 1 to 9 inclusive, such that in the final solution any given digit appears exactly once within its row, column, and subgrid. The original puzzle is mostly blank, with only some numbers pre-filled as hints. The player must use these hints to determine how to fill in the remaining squares through process of elimination, logical deduction, and trial and error. In the image above, the bold black numbers are the original hints given by the puzzle; the italic red numbers are those filled in by the player to produce the solution. In order to be a "proper" Sudoku puzzle, a given set of hints must have one unique solution.

The properties of Sudoku puzzles have lent it to a great deal of study by mathematicians and computer scientists. Considerable research has been put into finding the minimum number of clues that can be given while still producing a unique solution (17), and into finding puzzles that follow certain patterns. Solving Sudoku puzzles efficiently is a somewhat difficult task in computer science; it falls into the category of problems known as "NP-complete." This means that it is believed that no algorithm exists that can solve a Sudoku puzzle in less than polynominal time (without having loops nested at least two deep).

Problem Description

You will need to write a program that can read a Sudoku puzzle and find its solution. Remember, to solve a Sudoku puzzle, you must fill in all the blank squares with a

number between 1 and 9 inclusive, such that each number appears exactly once in each row, column, and 3-by-3 subgrid.

All of the puzzles your program will be given will be “proper” Sudoku puzzles; as stated above, this means that each puzzle will have exactly one valid solution.

Sample Input

The first line of your program’s input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include nine lines of text. Each line will contain only the digits from 1 through 9 inclusive and underscores (_). Underscores represent blank spaces in the puzzle that must be filled. Digits represent hints that should remain in place in the final solution.

```
2
4_2_______
____6_____
_589______
_9__5__8_
___34__51

_________
__16__32_
_8_1______
3_62_9_4

_________
___16__52
_7_5_4_6
39_______
62____39_
_______6_
9___3_____
_5_71_94_
2_6_5_7
```

Sample Output

For each test case, your program must output the solved Sudoku puzzle, printing nine lines with nine digits per line.

```
462571839
913468572
758923146
194756283
827349651
```

635812497
541697328
289134765
376285914
568327419
439168752
172594836
394286175
625471398
781953264
947835621
856712943
213649587