2019

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.

2019 Problem Packet

Copies of each problem description will also be available on the contest website

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. The website will compile and run your code, and you will be notified if your answer is correct or incorrect.

Who is judging our answers?

We have a team of Lockheed Martin employees responsible for judging the contest, however most of the judging is done automatically by the contest website. The contest website will compile and run your code, then compare your program's output to the expected official output. If the outputs match exactly, your team will be given credit for answering the problem correctly. Our judging team will review the website's work, but in most cases the automated response will stand.

How is each problem scored?

Each problem is assigned a point value based on the difficulty of the problem. When the website runs your program, it will compare your program's output to the expected judging output. If the outputs match exactly, you will be given the points for the problem. There is no partial credit; your outputs must match *exactly*. If you are being told your answer is incorrect and you are sure it's not, double check the formatting of your output, and make sure you don't have any trailing whitespace or other unexpected characters.

We don't understand the problem. How can we get help?

If you are having trouble understanding a problem, you can submit questions to the problems team through the contest website. While we cannot give hints about how to solve a problem, we may be able to clarify points that are unclear. If the problems team notices an error with a problem during the contest, we will send out a notification to all teams as soon as possible.

Our program works with the sample input/output, but it keeps getting marked as incorrect! Why?

Please note that the official inputs and outputs used to judge your answers are MUCH larger than the sample inputs and outputs provided to you. These inputs and outputs cover a wider range of test cases. The problem description will describe the limits of these inputs and outputs, but your program must be able to accept and handle any test case that falls within those limits. All inputs and outputs have been thoroughly tested by our problems team, and do not contain any invalid inputs.

We can't figure out why our answer is incorrect. What are we doing wrong?

Common errors may include:

1. Incorrect formatting - Double check the sample output in the problem and make sure your program has the correct output format.
2. Incorrect rounding - See the next section for information on rounding decimals.
3. Invalid numbers - 0 (or 0.0, 0.00, etc.) is NOT a negative number. 0 may be an acceptable answer, but -0 is not.
4. Extra characters - Make sure there is no extra whitespace at the end of each line of your output. Trailing spaces are not a part of any problem's output.
5. Decimal format - We use the period (.) as the decimal mark for all numbers.

If these tips don't help, feel free to submit a question to the problems team through the contest website. We cannot give hints about how to solve problems, but may be able to provide more information about why your answers are being returned as incorrect.

Who writes these problems?

All of the problems you're solving were written and tested by Lockheed Martin employees. In many cases, these problems are similar to real problems we have to deal with every day working at Lockheed Martin. If you'd like to learn more, ask any volunteer for more information!

Can I get answers to these problems after the contest?

Certainly! A member of our problems team will be available after the contest to answer any questions you have (and go over any incorrect answers you submitted so you can see why they were wrong). If you'd like a copy of our solutions to these problems, or those submitted by other teams, ask your coach to send an email to our Global Problems Lead, Brett Reynolds, at brett.w.reynolds@lmco.com. We can also provide copies of the official inputs and outputs used to judge your solutions.

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)

Please note that these tiebreaker methods may not be fully reflected on the contest website's live scoreboard. Additionally, the contest scoreboard will "freeze" 30 minutes before the end of the contest, so keep working as hard as you can!

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

Author: Brett Reynolds, Orlando, Florida, United States

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

Author: Shelly Adamie, Fort Worth, Texas, United States

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

Author: Shelly Adamie, Fort Worth, Texas, United States

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: Caught Speeding

Points: 10

Author: Holly Norton, Fort Worth, Texas, United States

Problem Background

You are driving a little too fast, and a police officer pulls you over. He needs to determine how big your speeding ticket should be; fortunately for you, he's decided to give you a bit of a break if it happens to be your birthday.

Problem Description

Your program should compute the ticket you are going to receive based on the speed you were travelling:

- If your speed is 60 or less, you don't get a ticket.
- If your speed is between 61 and 80 inclusive, you get a small ticket.
- If your speed is 81 or higher, you get a big ticket.

If today is your birthday, all of these numbers are increased by 5 (for example, you can drive up to 65 without getting a ticket).

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 two values separated by spaces:

- A positive integer representing your speed
- The word "true", indicating today is your birthday, or the word "false", indicating it is not.

```
3
60 false
82 false
83 true
```

Sample Output

For each test case, your program must print a single line, as follows:

- Print “no ticket” if you do not receive a ticket
- Print “small ticket” if you receive a small ticket
- Print “big ticket” if you receive a big ticket

```
no ticket
big ticket
small ticket
```

Problem 5: Brick House

Points: 10

Author: Holly Norton, Fort Worth, Texas, United States

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 6: Around and Around

Points: 15

Author: Chris Mason, Sunnyvale, California, United States

Problem Background

In 1962, John Glenn completed a historic spaceflight, orbiting the Earth three times in a small spacecraft. This flight was one of many that paved the way for an era of space exploration, eventually leading to the moon landings just seven years later. Despite the seemingly simple nature of Glenn's flight, it still required very precise calculations to ensure that he remained in orbit and didn't either fly off into space or come crashing back to Earth.

Objects in orbit don't remain in space simply because they've left Earth's atmosphere; they're still constantly falling towards Earth. The reason they stay in space is because they're moving so fast that they continually "miss" the Earth as they fall. In the case of John Glenn's historic flight, he was moving at an orbital speed of 17,544 miles per hour (28,234.8 kilometers per hour). This is fast enough to travel from New York to London in less than 12 minutes. During his entire flight, which lasted just short of five hours, Glenn travelled a total distance of 75,679.3 miles (121,794 kilometers).

Your task today is to determine how far an object in orbit at a particular height will travel during a single orbit of Earth.

Problem Description

Your program will be given the altitude of an object orbiting around Earth at the equator. Using this information, your program must calculate the total distance travelled by that object during a single orbit. It will help you to know that the circumference of the Earth at the equator is 40,075 kilometers.

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 an integer representing the object's altitude above the Earth's sea level in kilometers. Altitudes will be greater than or equal to 160 (the lowest possible orbital height for Earth).

```
3
160
200
265
```

Sample Output

For each test case, your program must output the distance travelled by an object orbiting around the equator at the given altitude in kilometers. Each value should be rounded to the nearest tenth of a kilometer (one decimal place).

```
41080.3
41331.6
41740.0
```

Problem 7: Image Compression

Points: 15

Author: Steve Brailsford, Marietta, Georgia, United States

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 8: Foveated Rendering

Points: 15

Author: Gary Hoffmann, Denver, Colorado, United States

Problem Background

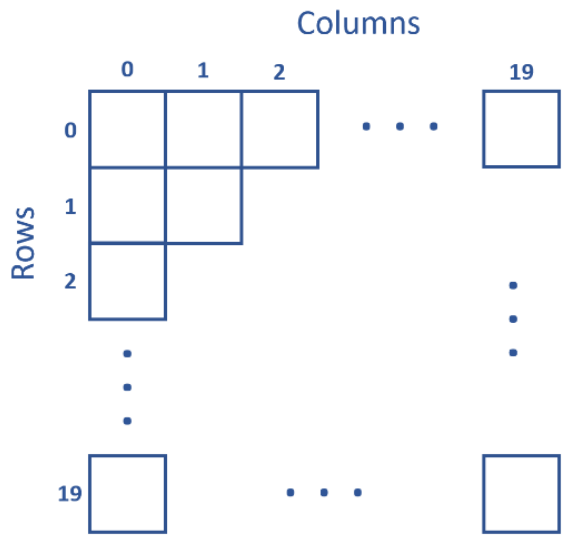
Virtual Reality has exploded into the market in the last five years, being used for everything from games and entertainment to product design and engineering. One of the more recent advances in VR headset design is the addition of eye tracking to increase performance.

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 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 25 25 25 25 10 10 10 10 10 10 10
```

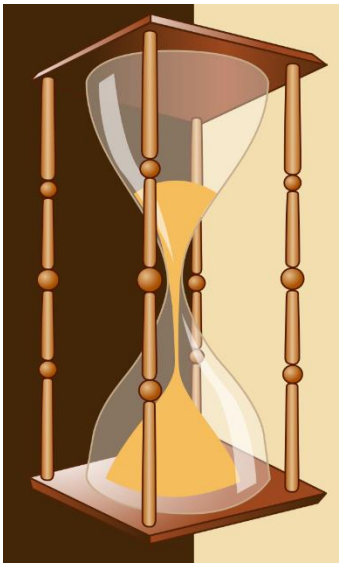
```
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 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 9: Time and Time Again

Points: 20

Author: Jonathan Brown, Fort Worth, Texas, United States

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.

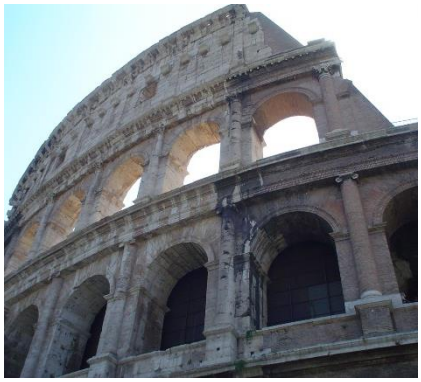
```
5
1m and 45s
10m,10s
32s, and 12h
76h
1s
```

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 10: Caesar Cipher



Points: 20

Author: Steve Gerali, Denver, Colorado, United States

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 11: Count to 10

Points: 25

Author: Ryan Regensburger, Huntsville, Alabama, United States

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 12: Monty Hall

Points: 25

Author: Christian Lin, Greenville, South Carolina, United States

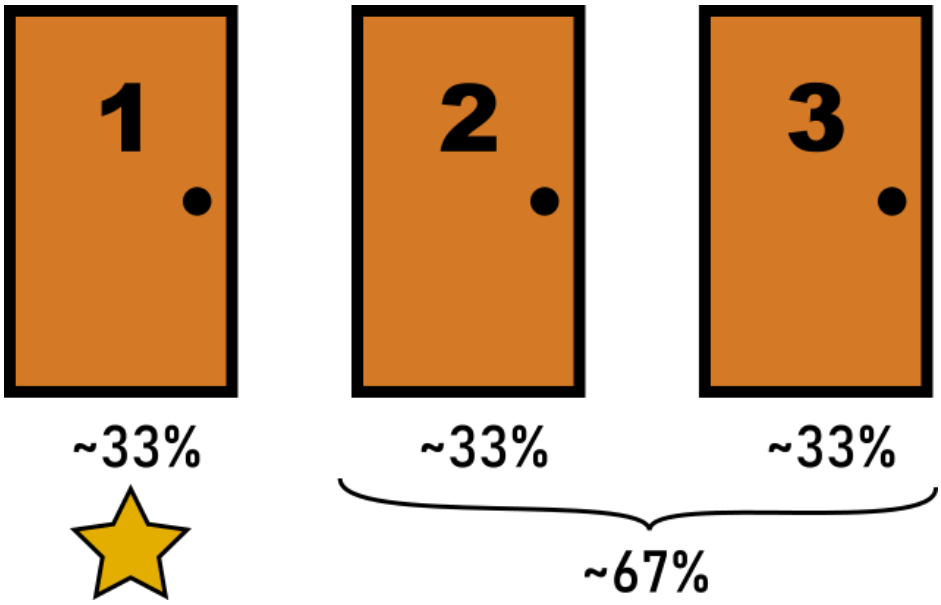
Problem Background

The “Monty Hall Problem” is a statistical problem named after the original host of the game show “Let’s Make a Deal!” In one of the game’s more famous segments, the host would give a contestant the choice of three doors. Behind one door was a car, but goats were behind the other two. The contestant would pick a door - for example, Door Number 1 - and the host, who knew what was behind each door, would pick a different door - for example, Door Number 3 - and open it. The host’s door would always contain a goat. The host would then give the contestant the option to switch to the other unopened door (Door Number 2, in this case).

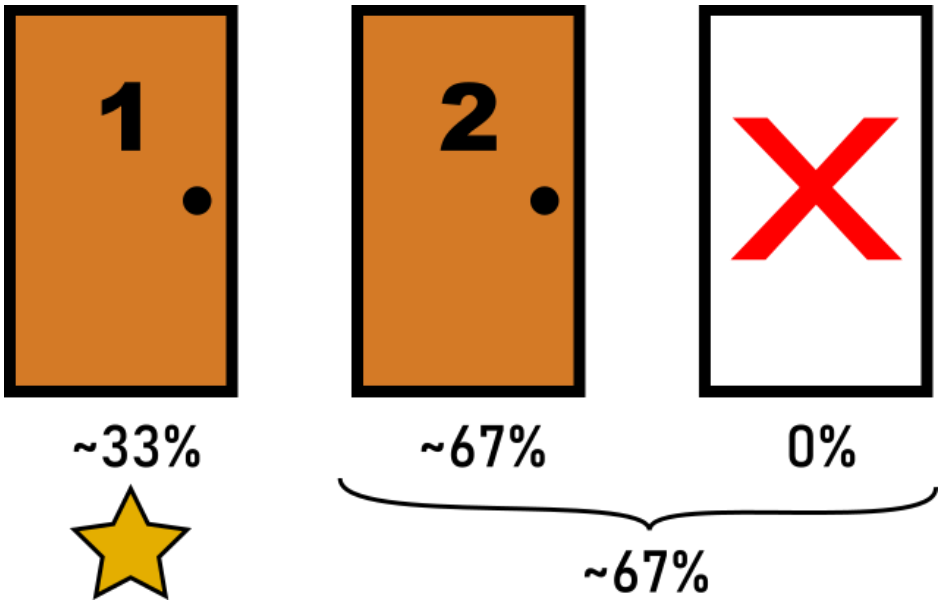
The problem is this: is it to the contestant’s advantage to switch doors? Does it matter if they switch?

It may seem counter-intuitive, but the answer is yes - the contestant is actually twice as likely to win if they switch! The reason why has stumped even experts in mathematics until they had the solution proved to them. Here’s the solution:

When the contestant makes their first choice, each door has a 33% chance of having a car behind it. To put this another way, there’s a 67% chance the contestant is wrong - that the car is behind one of the other two doors.



When the host opens one of the two incorrect doors, these probabilities don't change; the contestant's door still has a 33% chance of being right, and a 67% chance of being wrong. What *has* changed is that we know one of the other two doors in in fact wrong; it now has a 0% chance of being correct… shifting the 67% correct chance once shared between the two unselected doors to the single remaining door.



To summarize, there's still a 67% chance that the contestant is wrong, but with only one other option remaining, that means the contestant has a 67% of being *right* if they switch to that other option. It's not a guarantee, but they're still twice as likely to win!

Problem Description

You've been hired by a TV studio that wants to create a new game show based upon the Monty Hall problem. In case contestants are familiar with the problem, however, they're changing the game to make things more exciting.

The game will start with a number of doors (greater than 3). As before, only one door actually contains a prize. The contestant will pick one of these doors at the start of the game. The host will then open one or more of the doors the contestant did not select that do not contain prizes. The contestant will then be given the opportunity to select a new door if they wish. This process continues until the last round, where the contestant is given one final chance to switch doors. The door with the prize is then revealed.

The TV studio wants to conduct simulations of what they believe will be a worst-case scenario; a particularly knowledgeable contestant, and a particularly helpful host. Specifically, they want to run simulations in which the contestant and host follow these rules:

- When given the option to select a door, the contestant will select the door with the highest probability of winning. In the event of a tie, the contestant will select the door with the lowest number amongst those that are tied.
- When opening doors, the host will open the doors that had the highest probability of winning after the previous round. The host will never open the door with the prize and will never open any door the contestant has ever selected (either currently or previously).

For example, consider a game in which there are ten doors. The prize is behind door number 6 (marked in green), and three doors will be opened in each of two rounds. The contestant starts by selecting the lowest numbered door, 1 (marked in yellow)

10%	10%	10%	10%	10%	10%	10%	10%	10%	10%

The host now opens three doors. Since all doors have an equal chance of having the prize, he opens the last three doors: 8, 9, and 10. This increases the probability of all the doors the contestant did not select:

10%	15%	15%	15%	15%	15%	15%	X	X	X

The contestant is then given a chance to switch doors. The unselected, unopened doors now each have a 15% chance of being correct, so the contestant selects the first one of them. The host then opens three more doors with a high probability of winning. Normally he would open doors 5, 6, and 7, but door 6 contains the prize. As a result, he skips over door 6 and opens doors 4, 5, and 7 instead:

10%	15%	37.5%	X	X	37.5%	X	X	X	X

Only four doors remain now, and the contestant is again given the chance to switch. With two doors now at a 37.5% probability of being correct, he selects the first one of those. Unfortunately, his choice is incorrect, but he's still more than tripled his chances of winning from when the game started, simply by switching doors.

Your task is to write a program that will simulate several variations of this game. In each simulation, the contestant and host will both follow the rules outlined above. You must determine what the contestant's chances of winning was are the end of each game.

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 three positive integers, separated by spaces. These integers represent, in order:

- The number of doors at the start of the game
- The number of rounds during which doors will be opened by the host
- The number of doors opened in each round

```
3
10 2 3
10 3 2
10 4 1
```

Sample Output

For each test case, your program must output a single line containing the probability that the contestant will win the prize at the end of the game, assuming both contestant and host follow the rules outlined above. Probabilities should be printed as a percentage rounded to two decimal places (include any trailing zeroes).

```
37.50%
57.86%
24.61%
```

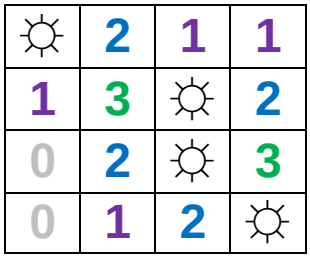
Problem 13: Minesweeper

Points: 25

Author: Lourdes Tuma, Denver, Colorado, United States

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 14: Homeward Bound

Points: 30

Author: Steve Brailsford, Marietta, Georgia, United States

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 15: LMCoin

Points: 35

Author: Ben Fenton, Faslane, Helensburgh, United Kingdom

Problem Background

Bitcoin and other "cryptocurrencies" are fast becoming very popular in today's world. They are vastly different from traditional ways of paying for things, in that they don't' require a bank or credit card company to act as a mediator between the buyer and seller. This avoids having to give the bank or other organization a transaction fee.

Instead, bitcoin is given directly to the other party. However, this leaves a big problem - how do you prove you've paid for something? Or that you even have the money to pay in the first place without someone vouching for you? This is known as the "double spending problem."

Instead of a bank recording all transactions in a central ledger, all users of bitcoin record all of the transactions at the same time. This means that any attempt to fool the system would be noticed and the transaction rejected. This is done through something called "blockchain." In this problem, we will build a simple blockchain for LMCoin, our very own digital currency.

As the name implies, a blockchain consists of several "blocks" of data, each representing a separate transaction. Each block is identified by a unique "hash", a value that is generated using all of the information stored in a block; this includes the hash of the previous block in the chain. Therefore, as we go along the chain, the integrity of the chain increases and ensures past transactions cannot be altered. Bitcoin and other cryptocurrencies are created by "mining" them; adding a new block to the chain that produces a unique "hash" value within a certain range - this involves a lot of guesswork!

Problem Description

You will need to write a program that implements the LMCoin blockchain. As the name implies, a blockchain consists of several "blocks" of data, each representing a separate transaction. Besides the start block (which doesn't include the last item), each block has four pieces of information:

1. A timestamp indicating when the block was created
2. Some data (such as a pizza order, for example)
3. An index (this block's position in the chain)
4. The hash of the previous block in the chain

The hash algorithm used by our currency will work as follows:

$$H_n = \frac{(T_n + V_n + n + H_{n-1}) * 50}{147}$$

Where:

- n is the index of the block within the chain; the first block in the chain has index 1.
- H_n is the hash for the block at index n. ($H_0 = 0$)
- T_n is the timestamp of the block at index n.
- V_n is the numeric value of the data in the block at index n (explained below).

A block's data is converted to a numeric value by adding the values of each letter in the data string, as shown below:

Letter	a	b	c	d	e	f	g	h	i	j	k	l	m
Value	1	2	3	4	5	6	7	8	9	10	11	12	13
Letter	n	o	p	q	r	s	t	u	v	w	x	y	z
Value	14	15	16	17	18	19	20	21	22	23	24	25	26

For example, the value of the string "cheese" is 3 + 8 + 5 + 5 + 19 + 5 = 45.

Timestamps will be in the format DDMMYYHHMM (two digits each for day, month, year, hour, and minute, respectively); for example, 8:30 AM on 27 April 2019 would be written as 2704190830.

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 space-separated list of data values for the first ten blocks in a chain. Data values will contain only lowercase letters.
- A line containing a space-separated list of timestamps for the first ten blocks in a chain. Timestamps will be in the format described above.

```
2
pepperoni veggie ham peppers cheese olives mushroom chicken beef bacon
2704191000 2704191030 2704191100 2704191130 2704191200 2704191230
2704191300 2704191330 2704191400 2704191430
candy candy salad chips pretzel icecream apple fries cookie sandwich
2602201200 2602201300 2702201200 2702201300 2802201200 2802201300
2902201200 2902201300 0103201200 0103201300
```

(Note that there are only four lines of input above following the number of test cases; the timestamps are too long to fit on this page.)

Sample Output

For each test case, your program must output the hash of the tenth block in the chain, calculated using the provided values. Printed hash values should be rounded to the nearest whole number. Do not round any intermediate hash values used for calculations.

```
1393884230
219309065
```

Problem 16: Have You Seen My Key

Points: 35

Author: Marcus Garza, Fort Worth, Texas, United States

Problem Background

In cryptography, there is an encryption scheme which is assumed to be perfect (unbreakable) under some key assumptions, called the one-time pad (OTP) cipher. The premise behind this cipher is that each encryption key is at least as long as the message itself, and once used, is never used again.

Two weaknesses in the OTP scheme are the probability of a codebreaker knowing the type of information being encrypted (the "lexicon") and the size of the key space. However, even if a codebreaker knows the lexicon, a brute force attack - testing every possible key - is essentially impossible.

Let's assume you're trying to encrypt a message that has a key that is 2^{512} bits long. 2^{512} is a BIG number:

13,407,807,929,942,597,099,574,024,998,205,846,127,479,365,820,592,393,377,723,561,443,721,764,030,073,546,976,801,874,298,166,903,427,690,031,858,186,486,050,853,753,882,811,946,569,946,433,649,006,084,096

If a codebreaker tried a brute-force attack against this message, and was able to test one key every nanosecond (0.000000001 seconds), it would take 100 trillion trillion trillion trillion trillion trillion trillion trillion trillion years (that's 9 "trillions") to test every key. That's several orders of magnitude greater than the age of the universe, and while it would eventually give you the correct plaintext, it would also give you every other potentially correct plaintext, making it impossible to determine which message was *actually* correct.

Problem Description

Your program will provide an implementation of the one-time pad cipher. Your program will be given a 128-character hexadecimal string representing the encrypted ciphertext. You will then be given another 128-character hexadecimal string representing the key. The plaintext consists of 64 ASCII characters. To convert the ciphertext to the plaintext, use this process, illustrated in the table below.

1. Get the next two hexadecimal characters from the ciphertext (e.g. '4F')
2. Convert this hexadecimal value to its 8-bit binary equivalent (4F = 01001111)
3. Get the next two hexadecimal characters from the key string (e.g. '0C')
4. Convert this hexadecimal value to its 8-bit binary equivalent (0C = 00001100)
5. XOR (exclusive-or) these two values together to create a new binary number

6. Convert the new binary number to an ASCII decimal value and print that character
7. Repeat with the rest of the message

Hexadecimal 4F = Decimal 79 = Binary 01001111

Hexadecimal 0C = Decimal 12 = Binary 00001100

4F	0	1	0	0	1	1	1	1
0C	0	0	0	0	1	1	0	0
XOR	0	1	0	0	0	0	1	1

Binary 01000011 = Decimal 67 = ASCII 'C'

If you are not familiar with XOR, it is a logical operation that checks to see if two boolean values are different. If they are, the result is true (1). If they are the same, the result is false (0).

XOR	0	1
0	0	1
1	1	0

Most programming languages allow you to XOR two numbers together to produce a new number; these "bitwise" operations perform an XOR comparison on each bit of the binary representation of those numbers, just as we did above in the example: 79 XOR 12 = 67. This can be done with the caret (^) operator in Java, C, C++, and Python, and the `Xor` operator in VB.NET.

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:

- A line containing a positive integer, **X**, representing the number of keys to use
- A line containing a 128-character hexadecimal string representing the ciphertext
- **X** lines, each containing a 128-character hexadecimal string representing a key

```
1
2
4F6F0E14089E040E28615606189340 4F658D1F6510D5744098DB1DF8904D5F0DF23710
D30230F4F985D4FAAE50F4984AF40B4C98F70E98F94998F043DF16D89F
0C006A7128CF716B5B15766F6BB3263A0BAC3F227FBA1060F4AE7E93B0393069934E31
F3515F988FE0F48EC63F87FD6A847923FA9B6BF58A68B8D063FF36F8BF
1B07676728EE686F410F226360E7602704FE3F1178B05433F9B678D8F3242F65974564
B67A44D49BF0A0DACF7090F12C926E3EFD997AB8922CE1DE639E5799DE
```

(Note that there are only three lines of hexadecimal text above; they're wrapping because they don't fit on the page)

Sample Output

For each test case, your program must output the 64-character plaintexts obtained by decrypting the ciphertext with each provided key, one per line. The plaintexts must be surrounded by brackets. The plaintexts may include trailing whitespace characters, which should be included within the brackets.

```
[Code Quest is fun! Good luck today! Solve those problems!       ]
[This plaintext has the same ciphertext but a different key. AAAA]
```

Problem 17: Conway's Game of Life

Points: 40

Author: Louis Ronat, Denver, Colorado, United States

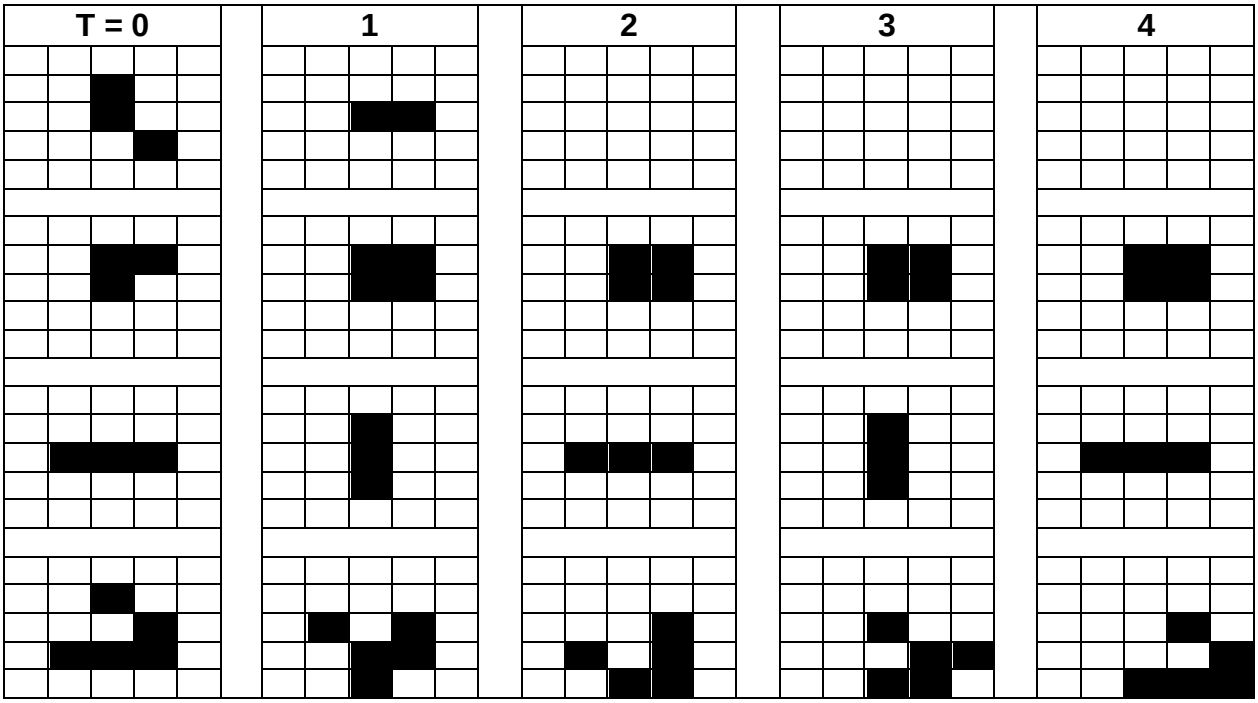
Problem Background

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 18: Mandelbrot Set

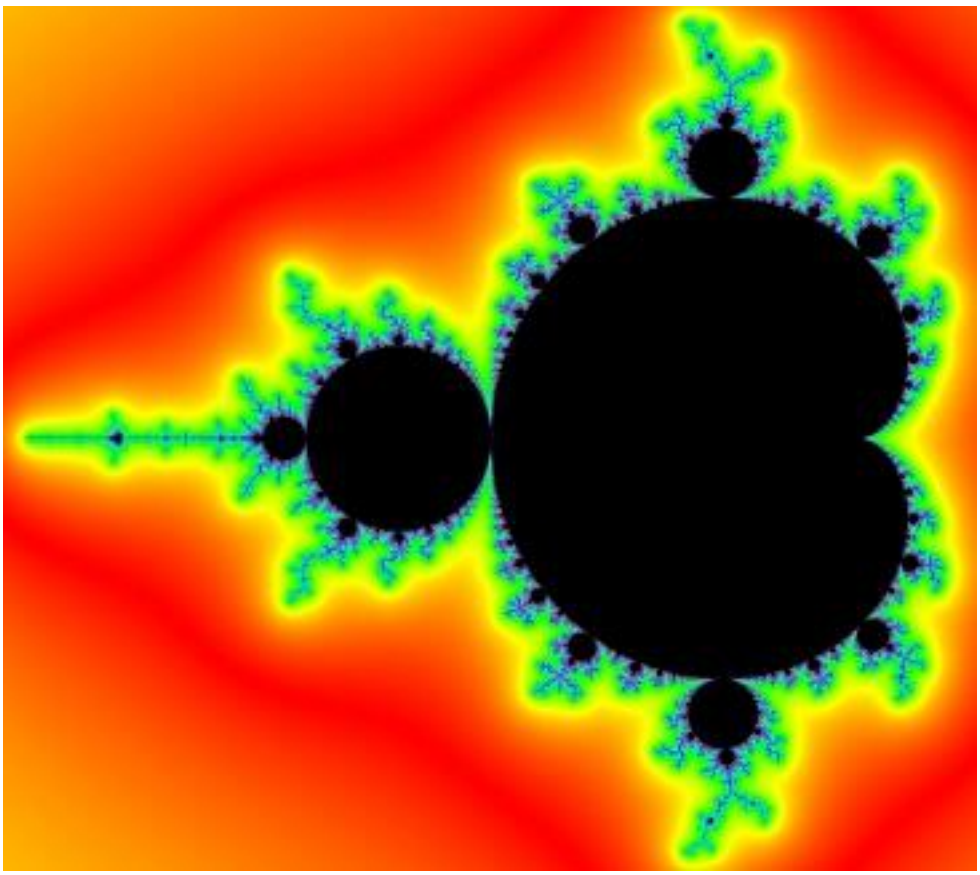
Points: 45

Author: Louis Ronat, Denver, Colorado, United States

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 19: Network Ranger

Points: 50

Author: Brett Reynolds, Orlando, Florida, United States

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	0	1	0	1	1	0	1	0	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 20: Bird Watching

Points: 55

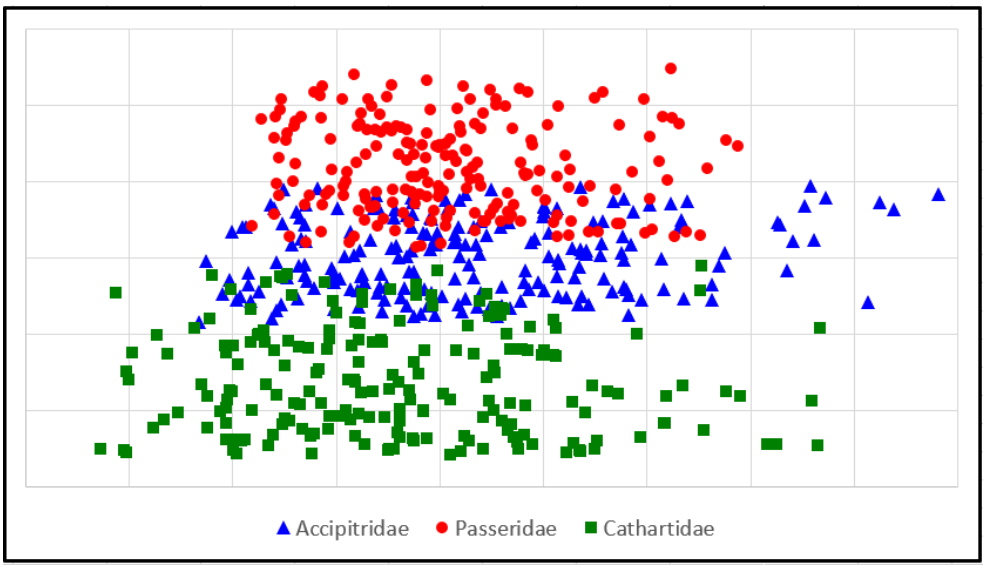
Author: Joe Worsham, Colorado Springs, Colorado, United States

Problem Background

Machine learning (ML) is a fancy type of artificial intelligence algorithm that uses patterns of previously-seen data to make predictions about new data that's never been seen before. ML algorithms can be very complex and can be used to solve extremely difficult problems; for example, neural networks simulate how individual cells in a real, living brain function. However, some are simple, straight-forward, and are great at solving pattern-related problems. In this problem, you'll need to create a machine learning system of your own!

You are working with the local park service to sort through pictures they've taken of birds recently. They want to be able to use these pictures to track the populations of certain species, but they have a very large number of photos and need to find a way to organize them. They decide to follow the practice of animal taxonomy and organize them by the family of the bird pictured in the photo. With the number of photos to sort through, they need an automated means of organizing the photos this way.

Your colleague suggests that you use measurements of the birds obtained from the photographs to predict a bird's taxonomic family. She retrieves a set of information about a long list of bird species and shows you there is an apparent correlation between certain measurements and a bird's family:



This graph is called a t-SNE embedding, a type of graph that converts multiple measurements (in this case four) into a two-dimensional coordinate. Each point is color-

coded based on the bird's taxonomic family. As your colleague points out, most of the colors are grouped together, giving her hypothesis that a pattern can be identified more weight. Her suggestion is to determine how "far away" an unknown bird is from these known measurements, and use that to make a reasonable prediction of the unknown bird's family.

Problem Description

Your colleague's idea is called a k-Nearest Neighbor (kNN) algorithm. This will predict an unknown bird's family based upon available taxonomic data and the measurements of the unknown bird. A kNN algorithm works by calculating the "distance" between an unknown datum point and each known data point. The *k* known data points closest to the unknown datum are then used to "vote" for the final decision.

In this problem, your algorithm should calculate the distance between the given unknown data point and all known data points. Once all distances have been calculated, count how many times each family of birds appears within the **K** closest points. This is the "voting" process mentioned before. Whichever family gets the most votes is selected as the family for the unknown bird.

In general, the value of **K** for this algorithm must be an integer; when there are only two possible answers, this is usually an odd integer, to avoid the likelihood of ties during the voting process. Here we have three possible answers, so ties will be possible. To address this, start with an initial value of **K** = 5 for all unknown birds. In the event there is a tie, increment **K** by 1 until the tie is broken; reset **K** to 5 for the next unknown bird.

The formula for calculating distance between *N*-dimensional points is as follows:

$$d_{p_1,p_2} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + \cdots + (N_1 - N_2)^2}$$

Each bird will be represented by four points of data in addition to its family - its length, body width, wingspan, and the angle of its wings relative to its body.

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 positive integers separated by spaces: **X**, representing the number of known birds, and **Y**, representing the number of unknown birds.
- A total of **X** lines containing information about known birds. Each line will contain the following values, separated by spaces:
    - One of the words "Accipitridae", "Passeridae", or "Cathartidae", representing the bird's taxonomic family
    - A decimal number representing the bird's length in inches.

    - A decimal number representing the bird's body width in inches.
    - A decimal number representing the bird's wingspan in inches.
    - A decimal number representing the bird's wing angle in degrees.
- A total of **Y** lines containing information about unknown birds in your available photographs. Each line will contain the following value, separated by spaces:
    - A decimal number representing the bird's length in inches.
    - A decimal number representing the bird's body width in inches.
    - A decimal number representing the bird's wingspan in inches.
    - A decimal number representing the bird's wing angle in degrees.

```
1
15 3
Accipitridae 12.30 7.03 25.32 88.59
Accipitridae 21.38 7.57 22.18 88.71
Passeridae 16.57 7.05 25.88 89.27
Passeridae 13.34 6.24 21.37 88.95
Passeridae 15.75 6.58 22.16 89.35
Accipitridae 15.16 5.17 22.43 89.04
Cathartidae 18.61 6.68 23.37 88.83
Accipitridae 21.32 8.14 20.09 88.55
Cathartidae 18.35 7.01 20.64 88.14
Cathartidae 13.61 5.33 23.72 90.21
Cathartidae 16.88 6.63 24.59 88.48
Accipitridae 15.63 8.66 23.19 88.51
Passeridae 17.29 7.62 26.46 89.31
Passeridae 20.03 8.68 20.97 89.05
Cathartidae 19.19 7.74 22.31 88.09
19.37 15.35 17.30 15.28
12.76 21.96 14.41 16.84
20.33 15.51 16.29 17.10
```

Sample Output

For each test case, your program must print the predicted taxonomic family for each unknown bird, one per line.

```
Accipitridae
Cathartidae
Accipitridae
```

Problem 21: Hide Your Spies

Points: 55

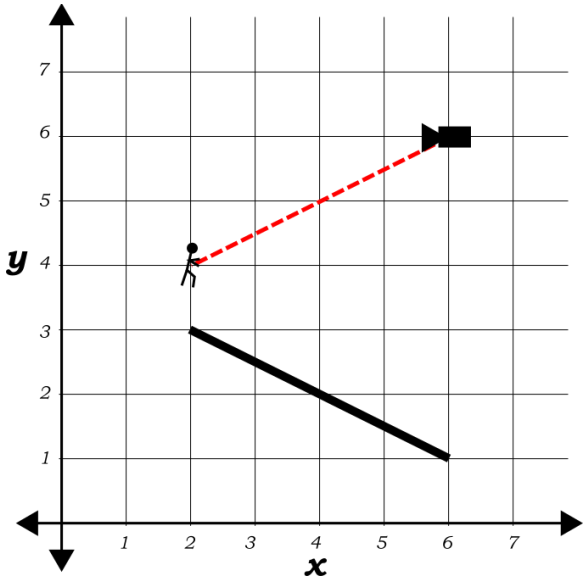
Author: Wojciech Kozioł, Mielec, Poland

Problem Background

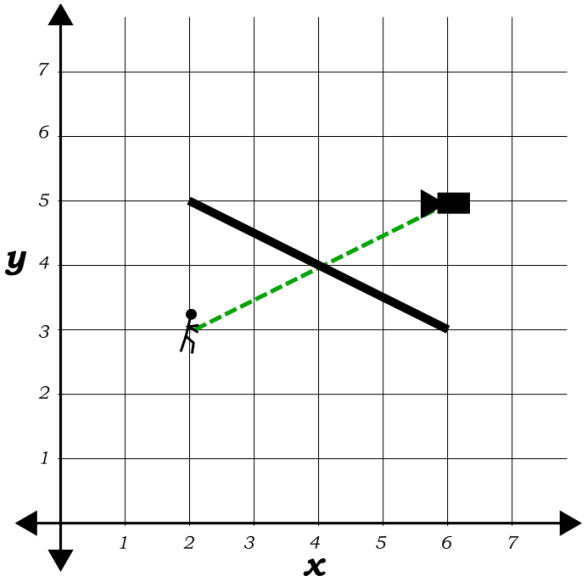
You're working with an intelligence agency to guide a spy through a secret enemy installation. The enemy has cameras positioned throughout the building with a 360° field of view; if your spy is caught on camera, the mission will fail! Fortunately, there are a number of walls blocking the view of the 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 22: Free Up Disk Space

Points: 60

Author: Doug Kelley, Palmdale, California, United States

Problem Background

The amount of disk space on your computer is getting low. We need an algorithm to archive the oldest files (because they probably aren't being used any more) and the biggest files (because they take up the most space).

Remember that a kilobyte (KB) is 1,000 bytes (B). A megabyte (MB) is 1,000 KB. A gigabyte (GB) is 1,000 MB.

Problem Description

Your program will be given a list of files on your computer and some information about them. Each file should be assigned a score based on its age and size. Using today's date of April 27th, 2019, determine the age of the file in days. A file created in the morning (from 12:00 AM through 11:59 AM) should be counted as ½ day older than one created the same day in the afternoon (12:00 PM through 11:59 PM); thus, a file last modified yesterday at 1:00 PM is 0.5 days old; one modified yesterday morning is 1.0 days old. Multiply the file's age in days by its size in MB to determine the file's score. Remember to account for leap years in your calculations.

For example, a 1500 KB file was last modified on April 27, 2018, at 10:00 PM. The file is 364.5 days old (365 days, minus 0.5 days for an afternoon time). Multiplying this value by the file's size in MB - 1500 KB = 1.5 MB - results in a score of 546.75.

Your program must list the highest-scoring files and their scores until the total size of the listed files accounts for at least 25% of your hard drive's capacity.

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, **F**, indicating the number of files on your computer, a space, and a positive decimal, **C**, indicating the size of your hard drive in GB.
- **F** lines listing the following information. Each data point is separated by spaces.
    - The date the file was last modified, in DD/MM/YYYY format. All dates will be no later than April 26, 2019.
    - The time the file was last modified, in HH:MM format

- "AM" or "PM", indicating if the timestamp was in the morning or afternoon
- A positive integer representing the size of the file in KB
- The name of the file. File names may include uppercase and lowercase letters, numbers, and periods (.).

```
1
10 1.0
25/04/2019 10:30 AM 125000 file1.txt
02/03/2019 02:15 PM 62500 file2.exe
01/01/2019 05:34 PM 62500 file3.mov
31/12/2018 11:36 AM 31250 file4.gif
14/02/2019 10:42 PM 31250 file5.doc
23/08/2018 12:00 PM 31250 file6.sh
29/02/2016 09:20 AM 31250 file7.mp3
05/12/2018 01:30 PM 15625 file8
26/04/2019 01:30 PM 15625 file9.png
01/01/2000 04:15 PM 1000 file10.jpg
```

Sample Output

For each test case, your program must output the information listed below for the highest-scoring files. Continue listing files until the total size of the listed files is equal to or greater than 25% of **C**.

- The name of the file
- A space
- The score calculated for that file, rounded to three decimal places. Include any trailing zeroes.

```
file7.mp3 36031.250
file6.sh 7703.125
file3.mov 7218.750
file10.jpg 7055.500
file4.gif 3656.250
file2.exe 3468.750
file5.doc 2234.375
```

Problem 23: Evacuate!

Points: 70

Author: Richard Green, Whiteley, Hampshire, United Kingdom

Problem Background

It's your first day working at Lockheed Martin 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! (Quick note: This could happen! Ask a volunteer about Lockheed Martin's Destination Innovation program.)

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 24: Sudoku

Points: 80

Author: Brett Reynolds, Orlando, Florida, United States

Problem Background

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

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