Code Quest Problem Packet

2021 Annual International Contest

Saturday, April 24, 2021

Table of Contents

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.

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:

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

I get an error when submitting my solution.

When submitting a solution, only select the source code for your program (depending on your language, this may include .java, .c, .h, .cpp, .py, or .vb files). Make sure to submit all files that are required to compile and run your program. Finally, make sure that the names of the files do not contain spaces or other non-alphanumeric characters (e.g. "Prob01.java" is ok, but "Prob 01.java" and "Bob'sSolution.java" are not).

Can I get solutions to the problems after the contest?

Yes! Please ask your coach to email Code Quest® Problems Lead Brett Reynolds at brett.w.reynolds@lmco.com .

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!

Mathematical Information

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:

- 1.49 rounds down to 1
- 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:

- 1.5 rounds up to 2
- -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!

Trigonometry

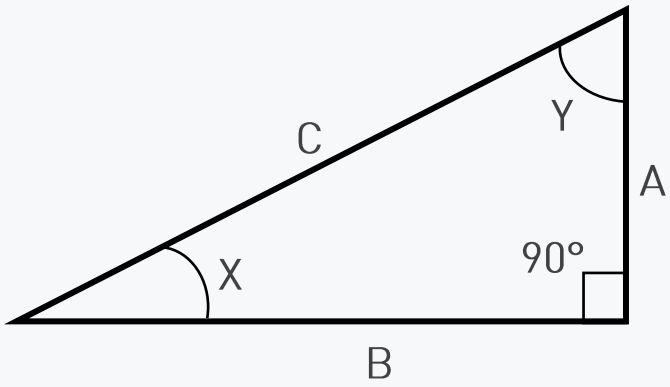
Some problems may require the use of trigonometric functions, which are summarized below. Most programming languages provide built-in functions for $\sin X$, $\cos X$, and $\tan X$; consult your language's documentation for full details. Unless otherwise stated in a problem description, it is *strongly recommended* that you use your language's built-in value for pi (π) whenever necessary.

$$\sin X = \frac{A}{C} \quad \cos X = \frac{B}{C} \quad \tan X = \frac{A}{B} = \frac{\sin X}{\cos X}$$

$$X + Y = 90°$$

$$A^2 + B^2 = C^2$$

$$\frac{degrees * \pi}{180} = radians$$



US ASCII Table

The inputs for all Code Quest® problems make use of printable US ASCII characters. Non-printable or control characters will not be used in any problem unless explicitly noted otherwise within the problem description. In some cases, you may be asked to convert characters to or from their numeric equivalents, shown in the table below.

Binary	Decimal	Character	Binary	Decimal	Character	Binary	Decimal	Character
0100000	32	(space)	1000000	64	@	1100000	96	`
0100001	33	!	1000001	65	A	1100001	97	a
0100010	34	"	1000010	66	B	1100010	98	b
0100011	35	#	1000011	67	C	1100011	99	c
0100100	36	$	1000100	68	D	1100100	100	d
0100101	37	%	1000101	69	E	1100101	101	e
0100110	38	&	1000110	70	F	1100110	102	f
0100111	39	'	1000111	71	G	1100111	103	g
0101000	40	(	1001000	72	H	1101000	104	h
0101001	41	)	1001001	73	I	1101001	105	i
0101010	42	*	1001010	74	J	1101010	106	j
0101011	43	+	1001011	75	K	1101011	107	k
0101100	44	,	1001100	76	L	1101100	108	l
0101101	45	-	1001101	77	M	1101101	109	m
0101110	46	.	1001110	78	N	1101110	110	n
0101111	47	/	1001111	79	O	1101111	111	o
0110000	48	0	1010000	80	P	1110000	112	p
0110001	49	1	1010001	81	Q	1110001	113	q
0110010	50	2	1010010	82	R	1110010	114	r
0110011	51	3	1010011	83	S	1110011	115	s
0110100	52	4	1010100	84	T	1110100	116	t
0110101	53	5	1010101	85	U	1110101	117	u
0110110	54	6	1010110	86	V	1110110	118	v
0110111	55	7	1010111	87	W	1110111	119	w
0111000	56	8	1011000	88	X	1111000	120	x
0111001	57	9	1011001	89	Y	1111001	121	y
0111010	58	:	1011010	90	Z	1111010	122	z
0111011	59	;	1011011	91	[	1111011	123	{
0111100	60	<	1011100	92	\	1111100	124	|
0111101	61	=	1011101	93	]	1111101	125	}
0111110	62	>	1011110	94	^	1111110	126	~
0111111	63	?	1011111	95	_			

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.

- An **integer** is any whole number; that is, a number with no decimal or fractional component: -5, 0, 5, and 123456789 are all integers.
- 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.
- **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.
- 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.
- **Positive numbers** are those numbers strictly greater than 0. 1 is the smallest positive integer; 0.000000000001 is a very small positive decimal number.
- **Non-positive numbers** are all numbers that are not positive; that is, all numbers less than or equal to 0.
- **Negative numbers** are those numbers strictly less than 0. -1 is the greatest negative integer; -0.000000000001 is a very large negative decimal number.
- **Non-negative numbers** are all numbers that are not negative; that is, all numbers greater than or equal to 0.
- **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.
- **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.
- **Date and time formats** are expressed using letters in place of numbers:
    - **HH** indicates the hours, written with two digits (with a leading zero if 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 (with a leading zero if needed; January is 01).
    - **YY** or **YYYY** is the year, written with two or four digits (with a leading zero if needed).
    - **DD** is the date of the month, written with two digits (with a leading zero if needed).

Problem 01: That's Odd

Points: 5

Author: Vedant Patel, Stratford, Connecticut, United States

Problem Background

Checking to see if a number is even or odd is a surprisingly common task in computer programming. For example, languages that do not provide Boolean data types can use even or odd integers in place of true or false.

Problem Description

Write a program that can determine if a number is even or odd. Even numbers are integers that are evenly divisible by 2 - that is, no remainder is left after the division. Any integer that is not even is odd.

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 an integer.

```
3
0
1
2
```

Sample Output

For each test case, your program must print a single line with the word "EVEN" if the given number if even, or "ODD" if the given number is odd.

```
EVEN
ODD
EVEN
```

Problem 02: Bigger Is Better

Points: 5

Author: Vedant Patel, Stratford, Connecticut, United States

Problem Background

Welcome to Lockheed Martin's Code Quest! We're glad you're here, because we could actually use your help. As you know, whichever team scores the most points in this contest will win... but there are a LOT of teams, and we could use some help identifying the highest score. If only there were a group of people who know how to write computer programs that could help us...

Problem Description

Your team will be given a list of numbers representing scores from Code Quest teams, and must identify the highest score in the list. Be careful, however; we warned you there are a lot of teams!

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 a list of integers representing team scores, each separated by spaces.

```
3
1 2 3 4
30 15 20 10
55 10 45 60 15 45 25 30
```

Sample Output

For each test case, your program must print a line containing the highest integer (closest to positive infinity) present in the given list.

```
4
30
60
```

Problem 03: Go for Two?

Points: 10

Author: Mark Rodrigues, Moorestown, New Jersey, United States

Problem Background

American Football is a hugely popular sport in the United States. Major companies often sponsor major tournament games called "bowl games," such as the Lockheed Martin Armed Forces Bowl. With as much attention as these games bring, there's a lot of pressure on coaches and players to perform at their best.

This pressure has increased the prevalence of sports analytics - the collection and analysis of historical statistics to help a team gain a competitive advantage over their opponent. Professional teams often have entire staffs dedicated to analyzing data and helping coaches make decisions based on that information.

Problem Description

One example of how analytics can be applied to a game is in making the decision to go for a one-point or two-point conversion after scoring a touchdown in American Football, particularly in the fourth quarter of the game. After scoring a touchdown, teams have the option to attempt to score an extra point by kicking the ball between the goal posts at the end of the field, or an extra two points by carrying the ball past the opposing team into the end zone.

Your team of analysts is working with a college competing in the Lockheed Martin Armed Forces Bowl to advise the coach when to call for a two-point conversion during the game's fourth quarter. Based on your team's and the opposing team's history, you've come up with this table of recommendations, based on how far ahead or behind your team is compared to their opponents.

Margin	Decision		Margin	Decision		Margin	Decision
Down by 15	2		Down by 5	2		Up by 5	2
Down by 14	1		Down by 4	2		Up by 6	1
Down by 13	2		Down by 3	1		Up by 7	1
Down by 12	1		Down by 2	2		Up by 8	1
Down by 11	2		Down by 1	1		Up by 9	1
Down by 10	2		Tied	1		Up by 10	1
Down by 9	1		Up by 1	2		Up by 11	1
Down by 8	2		Up by 2	1		Up by 12	2
Down by 7	1		Up by 3	1			
Down by 6	1		Up by 4	1			

For example, if your team is winning with 16 points to their opponent's 11 after scoring a touchdown, your team is up by 5 points and should try to make a two-point conversion. Any score margins not listed on the table should result in an attempt for a one-point conversion.

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 non-negative integers, separated by spaces. The first integer represents your team's score after they have scored a touchdown. The second integer represents the opposing team's score.

```
4
13 14
20 3
14 13
12 20
```

Sample Output

For each test case, your program must print a single line containing the number 1 or 2, indicating how many points your team should attempt to convert.

```
1
1
2
2
```

Problem 04: Phonebook

Points: 10

Author: Jonathan Tran, Dallas, Texas, United States

Problem Background

Phone numbers in the United States contain 10 digits; a 3-digit area code, a 3-digit exchange code, and a 4-digit line number. These three sections of a phone number are always separated, but the way in which they can be separated varies widely.

Problem Description

The three most common methods of writing phone numbers are:

- With parentheses: (317) 867-5309
- With dashes: 317-867-5309
- With periods: 317.867.5309

For this problem, you will be presented with a 10-digit number and must format it using one of the methods shown above.

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 two values separated by spaces:

- A ten-digit positive integer, representing the phone number to format
- One of the words "PARENTHESES", "DASHES", or "PERIODS", indicating one of the formats shown above.

```
3
1234567890 PARENTHESES
7531594862 DASHES
1470258369 PERIODS
```

Sample Output

For each test case, your program must print the phone number using the given format.

```
(123) 456-7890
753-159-4862
147.025.8369
```

Problem 05: Anti-Asteroid Weapon

Points: 15

Author: Luis Rivera, Aguadilla, Puerto Rico, United States

Problem Background

Earth is under attack... and you're our only hope!

A cluster of asteroids is quickly approaching Earth's atmosphere, and each of them is large enough to cause severe damage to all life on the planet. To protect us from this threat, world governments have worked with Lockheed Martin to develop a top-secret asteroid-destroying weapon to obliterate the space rocks without a trace.

The bad news is that the weapon takes about 20 minutes to charge between each shot... but the good news is that the asteroids are conveniently arriving in 20 minute intervals, so we should be able to shoot each one before it can impact us.

Problem Description

Your team needs to design a targeting algorithm to determine which of the asteroids needs to be shot first. The recharge delay on the weapon means that it's critically important that asteroids be destroyed in the correct order. Shooting an asteroid in the wrong order might allow another one to hit the Earth while the weapon is recharging.

NASA and the European Space Agency have been able to map the locations of each of the asteroids on a two-dimensional coordinate grid, with the Earth at the origin (0,0). You'll need to print out a list of the incoming asteroids in order of increasing distance from Earth. To calculate the distance from the origin point, you can use this formula:

$$d = \sqrt{x^2 + y^2}$$

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 a positive integer, A, representing the number of asteroids
- A lines containing two integers separated by spaces, representing the X and Y coordinates of each incoming asteroid, respectively.

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

Sample Output

For each test case, your program must print one line for each incoming asteroid, containing the coordinates of the asteroid as presented in the input. Asteroids must be printed in order by increasing distance from Earth.

```
1 1
2 1
5 5
-1 0
1 1
2 2
1 7
```

Problem 06: Dot Dot Dot

Points: 15

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

Problem Background

You've probably heard of Morse Code, in which each letter in the English alphabet is replaced with a series of dots and dashes. You and your friends were planning to use a clicker device to send secret messages to each other, but you realized that the devices only do one sort of click. You can make dots, but no dashes. Fortunately, you have a backup plan.

Problem Description

Your new code replaces each letter with a series of dots, equal to its position in the alphabet. For example, A gets replaced with one dot, B gets replaced with two, and so on, all the way to Z with 26 dots. Your thumbs are likely to get exhausted pressing the button on the clicker that many times, though... how many times will you need to click for each word?

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 with a single word written in lowercase letters.

```
4
code
quest
dot
problem
```

Sample Output

For each test case, your program must print the total number of dots (clicks) required to communicate the given word.

```
27
82
39
81
```

Problem 07: Enter the Matrix

Points: 20

Author: Dr. Francis Manning, Owego, New York, United States

Problem Background

In the fields of science, engineering, and technology, many products and projects that we deal with are highly dependent on specific mathematical operations to be able to function properly. Matrices are an excellent example of this; they have applications in computer graphics, aerospace engineering, and quantum mechanics. A matrix is a rectangular array of numbers, typically shown with two large square brackets:

$$M = \begin{bmatrix} m_{11} & \cdots & m_{n1} \\ \vdots & \ddots & \vdots \\ m_{1p} & \cdots & m_{np} \end{bmatrix}$$

Problem Description

Lockheed Martin is working with the United States Navy to test a new submersible drive system. The system involves two engines, which each have a different task in orienting the submarine. During the testing process, the submarine performs two maneuvers repeatedly, which place different demands on the two engines, as shown in this table:

Fuel Consumption	Maneuver 1	Maneuver 2
Engine 1	2 barrels	5 barrels
Engine 2	3 barrels	2 barrels

Based on the total amount of fuel consumption by each engine, we need to determine how many times each engine was fired. We can solve this problem using matrices.

We'll need three matrices to hold the information we need. First, our solution, the number of times each engine fired. This will be held in Matrix E:

$$E = [E_1 \quad E_2]$$

Second, the table above can be represented as Matrix C, showing how much fuel is consumed by each engine during each maneuver:

$$C = \begin{bmatrix} C_{1,1} & C_{2,1} \\ C_{1,2} & C_{2,2} \end{bmatrix} = \begin{bmatrix} 2 & 5 \\ 3 & 2 \end{bmatrix}$$

Finally, we need to know the total amount of fuel used during each maneuver. We'll keep this information in Matrix F.

$$F = [F_1 \quad F_2]$$

If we were dealing with a single engine and a single maneuver, multiplying the number of times the engine was fired by its fuel consumption rate would tell us the total amount of fuel used. The advantage of using matrices to hold all of this information is that this same logic holds true when working with multiple engines and maneuvers:

$$E \times C = F$$

$$[E_1 \quad E_2] \times \begin{bmatrix} 2 & 5 \\ 3 & 2 \end{bmatrix} = [F_1 \quad F_2]$$

Let's try an example. During one test, 49 barrels of fuel were spent on Maneuver 1 (F_1), and 73 barrels were used on Maneuver 2 (F_2). How many times was each engine fired?

In a normal algebra problem, we would divide F by C to determine the value of E. However, matrices can't be divided! Instead, we can invert a matrix, then multiply that inverse by a matrix to get the same result.

To calculate the inverse of a matrix, you can use the following formula:

$$X = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$X^{-1} = \frac{1}{(ad - bc)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \begin{bmatrix} \frac{d}{(ad - bc)} & \frac{-b}{(ad - bc)} \\ \frac{-c}{(ad - bc)} & \frac{a}{(ad - bc)} \end{bmatrix}$$

In short, we build a new matrix based on the values within the original; then each of those new values is divided by the original matrix's *determinant*. In order to be inverted, a matrix must be square, and it must have a non-zero determinant (since you can't divide by zero). For this problem, we will be inverting matrix C, which is square; in this example, C's determinant is $|C| = (2 \times 2) - (5 \times 3) = 4 - 15 = -11$, which is non-zero, so C can be inverted. For all test cases, C will have a non-zero determinant.

Once we invert C, we can multiply C^{-1} by F to determine the values in E. Using our example above:

$$C^{-1} = \begin{bmatrix} \frac{2}{-11} & \frac{-5}{-11} \\ \frac{-3}{-11} & \frac{2}{-11} \end{bmatrix} = \begin{bmatrix} -.1818 & .4545 \\ .2727 & -.1818 \end{bmatrix}$$

$$E = F \times C^{-1}$$

$$E = [49 \quad 73] \times \begin{bmatrix} -.1818 & .4545 \\ .2727 & -.1818 \end{bmatrix}$$

$$E = \left[\left({C^{-1}}_{1,1} F_1 + {C^{-1}}_{1,2} F_2 \right) \quad \left({C^{-1}}_{2,1} F_1 + {C^{-1}}_{2,2} F_2 \right) \right]$$

$$E = \begin{bmatrix} ((-.1818 \times 49) + (.2727 \times 73)) & ((.4545 \times 49) + (-.1818 \times 73)) \end{bmatrix}$$

$$E = \begin{bmatrix} 10.9989 & 8.9991 \end{bmatrix} \approx \begin{bmatrix} 11 & 9 \end{bmatrix}$$

And we have our answer! Since $E = \begin{bmatrix} 11 & 9 \end{bmatrix}$ (after rounding to the nearest whole number), we can see that engine 1 was fired 11 times, and engine 2 was fired 9 times during the course of the test.

In order to automate this testing routine, your team has been asked to write a program that can calculate the values in E given the values of C and F.

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 three lines, containing the information listed below. On each line, values are separated by spaces. All values will be positive integers.

- The first line will contain the values for $C_{1,1}$ and $C_{2,1}$; the fuel consumed by Engine 1 during Maneuver 1 and Maneuver 2, respectively.
- The second line will contain the values for $C_{1,2}$ and $C_{2,2}$; the fuel consumed by Engine 2 during Maneuver 1 and Maneuver 2, respectively.
- The third line will contain the values for F_1 and F_2; the total fuel consumed by both engines during Maneuver 1 and Maneuver 2, respectively.

```
2
2 5
3 2
49 73
7 3
4 5
193 155
```

Sample Output

For each test case, your program must print a single line containing the values for E_1 and E_2. These represent the number of times Engines 1 and 2 were fired during the test, respectively. Both values should be rounded to the nearest integer and be separated by spaces.

```
11 9
15 22
```

Problem 08: Normal Math

Points: 20

Author: Steve Brailsford, Marietta, Georgia, United States

Problem Background

Regardless of how advanced computer graphics are, it all boils down to a bunch of numbers. Graphics engines make use of large tables of numbers, called matrices, in order to keep track of every detail being rendered on your screen. Rotating, moving, or zooming into or out of an image can be represented by performing a number of mathematical operations on the matrix representing that image. However, matrix operations can be very complicated, and it's a good practice to check your work to ensure it is correct.

Lockheed Martin Aeronautics is trying to develop an augmented reality system to help train people to maintain the aircraft they create. However, the team working on the graphics overlays are having difficulty keeping things oriented the right way when the user moves their head. They'd like to add an error-detection system to help identify where the problem is coming from.

Problem Description

A computer graphics expert has suggested you calculate the Frobenius Norm of each matrix you create for use in the AR system. This is a number that can be used to identify when changes have taken place to a matrix. It's calculated using the formula below:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

...Or to put it in MUCH simpler terms, it's calculated by taking the absolute value of each element in the matrix, squaring them, adding them all together, then taking the square root of that sum. For example, let's consider a small 2x2 matrix:

$$A = \begin{bmatrix} 3 & -2 \\ 4 & 5 \end{bmatrix}$$

We can calculate the Frobenius Norm by taking the absolute value of each element...

$$|3| = 3 \quad |-2| = 2 \quad |4| = 4 \quad |5| = 5$$

...squaring those...

$$3^2 = 9 \quad 2^2 = 4 \quad 4^2 = 16 \quad 5^2 = 25$$

...adding all those squares together...

$$9 + 4 + 16 + 25 = 54$$

...then getting the square root of that.

$$\|A\|_F = \sqrt{54} \approx 7.35$$

Now why is this number useful? Because it's calculated using each of the individual values, any attempt to simply move the numbers in the matrix around - as might be done when rotating or translating it - will result in a matrix with the same Frobenius Norm. Scaling the values in a matrix (which might be done when zooming into or out of an image) will produce a different Frobenius Norm, but it will be equal to the original Frobenius Norm multiplied by the scale value.

$$A_R = \begin{bmatrix} 5 & 4 \\ = 2 & 3 \end{bmatrix}$$

$$\|A_R\|_F \approx 7.35$$

$$B = A * 4 = \begin{bmatrix} 12 & -8 \\ 16 & 20 \end{bmatrix}$$

$$\|B\|_F = \|A\|_F * 4 \approx 29.39$$

As a result of this, the Frobenius Norm can be used to quickly identify the presence of errors in matrix rotation, translation, or scaling. It may not tell you what the exact problem is, but it'll at least help you figure out where to start looking!

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 two positive integers, M and N, separated by spaces, representing the number of rows and columns (respectively) contained in a matrix. Values for both M and N will range from 1 to 30 inclusive.
- M lines, each containing N integers, separated by spaces, representing the content of the matrix.

```
4
2 2
3 -2
4 5
2 2
5 4
-2 3
2 2
12 -8
16 20
3 3
1 2 3
4 5 6
7 8 9
```

Sample Output

For each test case, your program must print the Frobenius Norm of the given matrix, rounded to two decimal places. Include any trailing zeroes.

```
7.35
7.35
29.39
16.88
```

Problem 09: Where's My Change?

Points: 20

Author: Steve Gerali, Denver, Colorado, United States

Problem Background

An automated teller machine (ATM) is an electronic banking device that allows customers to complete basic banking transactions like deposits or withdrawals without the aid of a human teller. Anyone with a credit or debit card can access most ATMs. The first ATM appeared in London in 1967; in just over 50 years, they have spread throughout the globe, with more than 3.5 million ATMs installed worldwide.

Problem Description

Your team is working with a bank that wants to deploy new ATMs throughout their banking network. While most ATMs will only dispense money in one denomination, the bank wants their new ATMs to be more flexible, dispensing cash in any legal denomination. They believe that this will allow their bank to dominate the ATM business by better addressing customer's needs.

Your team has been hired to develop the algorithm that determines how much change to dispense to each customer. The ATMs will be deployed in the United States, and so can dispense bills in denominations of $100, $50, $20, $10, $5, $2, and $1, and coins in denominations of $0.25, $0.10, $0.05, and $0.01. Your algorithm should direct the ATM to dispense the smallest number of individual bills and coins necessary to fulfill the customer's request. For example, if a customer requests $5.01, you should dispense one $5 bill and one $0.01 coin to meet that total. An order of $51.27 would require one $50 bill, one $1 bill, one $0.25 coin, and two $0.01 coins.

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 decimal value representing the amount of money a customer would like to withdraw from their bank account.

```
5
5.01
51.27
10.89
27.08
102.30
```

Sample Output

For each test case, your program must print a single line indicating the amount of each denomination of money to dispense to the customer. For each denomination, in the order listed below, print a single integer representing the amount of that denomination to dispense. You will never be required to dispense more than 9 items of a single denomination.

- $100
- $50
- $20
- $10
- $5
- $2
- $1
- $0.25
- $0.10
- $0.05
- $0.01

```
00001000001
01000011002
00010003104
00101100013
10000101010
```

Problem 10: Confusing Conversions

Points: 25

Author: Ryan Regensburger, Huntsville, Alabama, United States

Problem Background

A useful feature in many programming languages is the ability to create functions; sections of code that can be "called" on demand to perform a specific task, avoiding the need to rewrite the same algorithms over and over again. Functions are often given a set of "arguments," a collection of data presented in a specific order, which the function uses during its execution. Most functions "return" a value, which can be saved to a variable for later use or passed to another function as an argument. Collections of similar or related functions called libraries form the basis of many programming languages and allow developers to create powerful programs much more quickly.

Problem Description

Lockheed Martin's Enterprise Operations division is working to develop a set of libraries containing common utility functions, to be shared by all projects across the Lockheed Martin corporation. Your team has been assigned the library focused on data conversions. The quality control team has already created a series of tests to run against your new library; your job is simply to create the functions necessary to pass those tests. The functions you need to create include:

- formatHeight
  - Arguments:
    - X - an integer representing the number of feet
    - Y - an integer representing the number of inches
  - Returns a text string in the format X'Y" (the number of feet, an apostrophe, the number of inches, and a quote)

- formatDate
  - Arguments:
    - X - an integer representing the year
    - Y - an integer representing the month
    - Z - an integer representing the day of the month
  - Returns a text string containing the given date in YYYYMMDD format

- concatenate
    - Arguments:
        - X… - One or more text strings
    - Returns a text string containing all arguments in the order presented, with each argument separated from the next with a comma

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 the name of a function listed above, followed by the arguments for that function. All values will be separated by spaces.

```
4
formatHeight 5 8
formatDate 2020 5 2
concatenate These are all arguments
concatenate Text can be 123 numbers too
```

Sample Output

For each test case, your program must execute the named function with the provided arguments and print its return value on a single line.

```
5’8”
20200502
These,are,all,arguments
Text,can,be,123,numbers,too
```

Problem 11: Codebreaker

Points: 30

Author: Brett Reynolds, Annapolis Junction, Maryland, United States

Problem Background

A substitution cipher is a cipher that replaces letters (or sometimes groups of letters) with other letters (or groups), effectively scrambling a message and making it illegible. For example, a Caesar cipher, likely the first cipher ever created, works by "shifting" the alphabet. Each letter in the original "plaintext" message is replaced with another letter a certain number of positions further down the alphabet to create the encrypted "ciphertext." For example, in English with a shift of 3, A becomes D, B becomes E, and so on (X, Y, and Z wrap around to be replaced with A, B, and C respectively).

Plaintext: `ABCDEFGHIJKLMNOPQRSTUVWXYZ`

Caesar Shift 3: `DEFGHIJKLMNOPQRSTUVWXYZABC`

Substitution ciphers have a major vulnerability, however; some letters in the alphabet occur more frequently than others. Vowels in particular (in English, A, E, I, O, and U) are very common letters in any language, because there are so few of them and yet they are required to make words pronounceable. Certain consonants also occur more frequently than others (R, S, and T are much more common in English than Q, X, and Z). By counting how often certain letters occur within a message encrypted with a substitution cipher, a cryptanalyst (codebreaker) might be able to crack the cipher. By creating a chart showing the relative frequency of each letter in the ciphertext and comparing it against the average frequency of each letter in the expected language, a cryptanalyst can guess the substitutions used and begin to piece together the original message.

Problem Description

You're working on a new cybersecurity team within Lockheed Martin to help the National Security Agency develop a new frequency analysis database. You must write a program that is able to read in a large amount of text and count the number of occurrences of each letter of the English alphabet. It must then print the relative frequency of each number in a list, for use in future codebreaking endeavors.

The relative frequency of a letter is determined by the following formula, and is expressed as a percentage value:

$$Relative\ frequency = \left(\frac{Occurrences\ of\ letter}{Total\ number\ of\ letters}\right) \times 100\%$$

For example, in the sentence "The cow is brown," there are 13 letters. The letter 'o' appears twice, giving it a relative frequency of $(2 \div 13) \times 100\% = 15.38\%$. Punctuation and spaces should not be taken into account when calculating relative frequencies. Uppercase and lowercase versions of a letter should be counted as the same letter.

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 a positive integer, X, indicating the number of lines of text to be provided.
- X lines, containing text to be analyzed. Lines may contain up to 2000 characters each, and can contain any US ASCII character.

```
1
3
The quick red fox jumps over the lazy brown dog.
The above sentence contains every letter in the English language.
Don't forget to ignore punctuation and numbers; they're not relevant!
```

Sample Output

For each test case, your program must print a line for each letter in the English language, in order, using the following format:

- The letter, in uppercase
- A colon (:)
- A space
- The relative frequency of that letter within the text provided in the test case, rounded to two decimal places and including any trailing zeroes
- A percent sign (%)

```
A: 5.37%
B: 2.01%
C: 2.68%
D: 2.68%
E: 15.44%
F: 1.34%
G: 4.03%
H: 4.03%
I: 4.03%
J: 0.67%
K: 0.67%
L: 3.36%
M: 1.34%
N: 10.74%
O: 8.05%
P: 1.34%
Q: 0.67%
R: 6.71%
S: 3.36%
T: 10.74%
U: 4.03%
V: 2.68%
W: 0.67%
X: 0.67%
Y: 2.01%
Z: 0.67%
```

Problem 12: Complex Code

Points: 30

Author: Matt Hussey, Ampthill, Reddings Wood, United Kingdom

Problem Background

With a Code Quest problem, the main goal is to make sure your program gets the correct solution. It doesn't matter how clean your code is or (usually) how efficient it is; producing the correct output will get you points regardless.

In practice, however, software programs need to be maintainable. Writing neat code and creating detailed documentation helps others understand what your code does, which in turn makes it much easier for any developer to correct bugs in your code when they are found. Conversely, needlessly complicated code can be difficult to maintain, since it takes more time to figure out what the code is doing.

Problem Description

It's possible to analyze software code to determine its level of complexity. Two metrics used for this purpose are "cyclomatic complexity" and "nesting depth." Your team is working with systems engineers at Lockheed Martin to scan a section of code to determine if that code exceeds acceptable limits for these metrics.

Cyclomatic complexity is a measure of how many possible paths exist through a section of code. For example, consider the pseudocode below:

```
Print "Hello"
Print "World"
Print "This is not complex"
```

This code snippet contains three statements, but there's only one way to execute them; the program offers no alternative paths to take. It has a cyclomatic complexity of 1.

Now consider the snippet on the next page:

```
Print “I have a number”
If num > 2
{
      Print “It’s more than 2”
}
Else
{
      Print “It’s 2 or less”
}
```

In this snippet, the first print statement will always be executed. However, the program could execute either of the other two print statements, depending on the value of “num.” This creates two possible paths for the snippet to take, resulting in a cyclomatic complexity of 2.

Nesting depth indicates the maximum number of nested statements within a section of code. The first snippet above only included print statements; no nesting was involved. It has a nesting depth of 0. The second snippet included if and else statements. Each of those statements included a nested print statement. That snippet has a nesting depth of 1. The snippet below has a nesting depth of 3:

```
If num > 2
{
      If num > 3
      {
            If num > 4
            {
                  Print “It’s really big”
            }
            Else
            {
                  Print “It’s pretty big”
            }
      }
      Else
      {
            Print “It’s kinda big”
      }
}
Else
{
      Print “It’s not big”
}
```

The first two print statements are each nested within three layers of statements, resulting in the nesting depth of 3. Even though the code contains other nested statements, none are nested at a greater level than that, so they do not count towards the nesting depth. This snippet also has a cyclomatic complexity of 4, since there are four possible paths through the snippet;

one if num is 2 or less, a second if num is 3, a third if num is 4, and a fourth if num is greater than 4.

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 integer values separated by spaces:
  - L, a positive integer representing the number of lines of code to evaluate
  - C, a positive integer representing the highest acceptable cyclomatic complexity for the code snippet
  - N, a non-negative integer representing the highest acceptable nesting depth for the code snippet
- L lines containing a snippet of pseudocode, which will consist of a series of print, if, and else statements. Lines will not include indentation; the examples above were indented only to demonstrate the relevant concepts.
  - Print statements are a single line and will begin with the word "Print", which is then followed by a string of text wrapped in quotes (").
  - If statements cover multiple lines. The first line will start with the word "If", followed by a comparison. Comparisons will be made between a variable (containing only lowercase letters) and an integer number, and may use one of the operators >, <, >=, <=, ==, or !=. The next line will contain a left curly bracket ({). A later line will contain a matching right curly bracket (}), which terminates the if statement.
  - Else statements cover multiple lines, and will only occur immediately after the termination of an if statement. The first line will contain the word "Else". The next line will contain a left curly bracket ({). A later line will contain a matching right curly bracket (}), which terminates the else statement.

```
2
9 2 1
Print “I have a number”
If num > 2
{
Print “It’s more than 2”
}
Else
{
Print “It’s 2 or less”
}
11 2 2
If num > 3
{
If num > 4
{
Print “It’s really big”
}
Else
{
Print “It’s pretty big”
}
}
```

Sample Output

For each test case, your program must print a single line containing the following values, separated by spaces:

- An integer representing the snippet’s cyclomatic complexity
- An integer representing the snippet’s nesting depth
- If both of the above metrics were equal to or less than their limits (C and N, respectively), print the word “PASS”; otherwise, print “FAIL”

```
2 1 PASS
3 2 FAIL
```

Problem 13: Little Orphan ASCII

Points: 35

Authors: Cindy Gibson and Danny Lin, Greenville, South Carolina, United States

Problem Background

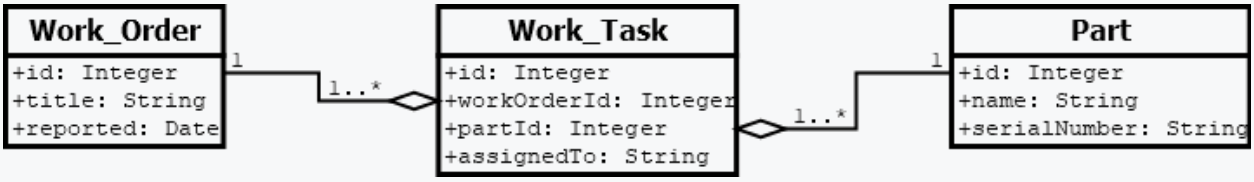
Databases are a critical part of many software systems. A database is sort of like a digital filing cabinet - it's a large repository for any sort of information your software program might need. As the program runs, it can add data to the database, modify it, or delete it. Many times, data records in your database will refer to other data records. For example, a record about a library book may contain a reference to the library from which it was checked out.

Maintaining these relationships between different parts of your data is called "data integrity." Most databases automatically manage data integrity - to use the example above, it doesn't make sense to say a book belongs to a library that doesn't exist. However, if a database isn't set up correctly, these relationships can break down, causing "orphaned data." Perhaps the library did exist once, but has since closed and thus been removed from the database. Unless their records are updated or deleted, any books in the library are now orphaned. We know they exist, but we have no idea where they belong. This can cause problems for a software application that expects every book to be neatly stored somewhere on a library shelf.

Problem Description

The US Army has contracted Lockheed Martin's Rotary and Missions Systems division to update its maintenance tracking software. Their existing system has become badly outdated, and wasn't set up with proper data integrity protections. Your task is to go through the old system's database and identify any records that have become orphaned so they can be reported to the Army and investigated.

The main area of concern is three tables in the database, illustrated below:



The Work_Order table contains information about problems that have been reported on the Army's equipment. Each work order contains one or more Work_Tasks, which detail the specific jobs that need to be completed to fix the reported problem. Each work task requires a Part from the supply in order to be completed. These relationships can be seen by the lines extending from the Work_Task table; the "workOrderId" field should contain a value matching

the "id" value of an entry in the Work_Order table, and "partId" should refer to a Part's "id" value.

Unfortunately, when the database was set up, these relationships were not clearly defined. As a result, over time the database has become corrupted, leaving orphaned task data that refers to a non-existent work order, a non-existent part, or both. Your task is to write a program that scans the database to identify these orphaned records, so the Army can search through their physical paperwork to rebuild their maintenance records.

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:
  - W, representing the number of work orders in the database
  - T, representing the number of work tasks in the database
  - P, representing the number of parts in the database
- W lines representing the data stored in the Work_Order table. Each line contains the following information, separated by commas (,):
  - A non-negative integer representing the ID value of the Work_Order. Each ID value will be unique amongst other Work_Order entries.
  - A text string (containing upper- and lower-case letters, spaces, and/or numbers) representing the title of the Work_Order
  - A date, in DD-MM-YYYY format, representing the date on which the Work_Order was reported.
- T lines representing the data stored in the Work_Task table. Each line contains the following information, separated by commas (,):
  - A non-negative integer representing the ID value of the Work_Task. Each ID value will be unique amongst other Work_Task entries.
  - A non-negative integer representing the ID value of the associated Work_Order. This number may not appear in the provided Work_Order entries.
  - A non-negative integer representing the ID value of the associated Part. This number may not appear in the provided Part entries.
  - A text string (containing upper- and lower-case letters and/or spaces) representing the name of the technician assigned to this Work_Task.
- P lines representing the data stored in the Part table. Each line contains the following information, separated by commas (,):
  - A non-negative integer representing the ID value of the Part. Each ID value will be unique amongst other Part entries.
  - A text string (containing upper- and lower-case letters, spaces, and/or numbers) representing the name of the Part.

- A text string (containing upper- and lower-case letters, dashes, and/or numbers) representing the serial number of the Part.

```
1
3 4 3
1,Dead Battery,29-04-2020
3,Flat Tire,01-05-2020
5,Door Jammed,02-05-2020
1,1,2,John Doe
2,2,4,Jane Doe
3,3,5,James Doe
4,4,7,Joan Doe
2,Truck Axle,12-34
4,Spare Tire,7890
6,Thingiemabob,3-14159
```

Sample Output

For each test case, your program must print information about each Work_Task entry that contains orphaned data. Entries with orphaned data should be printed in increasing numerical order by their ID value. Each line should include the following information, separated by spaces:

- The ID number of the orphaned Work_Task entry
- If the Work_Order ID is invalid, the phrase "MISSING WORK_ORDER ##", replacing ## with the invalid Work_Order ID
- If the Part ID is invalid, the phrase "MISSING PART ##", replacing ## with the invalid Part ID

```
2 MISSING WORK_ORDER 2
3 MISSING PART 5
4 MISSING WORK_ORDER 4 MISSING PART 7
```

Problem 14: Caesar With a Shift

Points: 40

Author: Carlos J. Sepulveda, Orlando, Florida, United States

Problem Background

A Caesar Cipher is a simple substitution cipher that replaces each letter in the alphabet with another letter according to a shift value. The most common shift value is 3, so A gets replaced with D, B with E, and so on, wrapping around so that XYZ becomes ABC:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
DEFGHIJKLMNOPQRSTUVWXYZABC
```

Unfortunately, the cipher is so common and easy to solve that it's not of much use... so let's make things a bit more interesting.

Problem Description

The Caesar Cipher we described above is a monoalphabetic cipher - there's only one replacement for each letter. With our variant, each letter will be encrypted differently based on a set of shift values and directions. Letters can be shifted between 1 and 25 positions, either left (backwards) or right (forwards).

For example, the letter A could be encrypted in several different ways:

- A right shift of 3 produces D, as it would in a traditional Caesar Cipher.
- A left shift of 3 produces X, wrapping around the end of the alphabet.
- A right shift of 10 produces K, the tenth letter in the alphabet after A.
- A left shift of 10 produces Q, ten letters before A (again, wrapping around).

The shifts and directions will be presented as sets of numbers; the shifts between 1 and 25 inclusive, and the directions either 0 (for left) or 1 (for right). The sets may not be the same length, and most likely will be shorter than the message you're trying to decrypt. Whenever you reach the end of a key set, loop around to the beginning of the set again.

Given an encrypted message and the shift and directions sets used to encrypt it, your team will need to reverse the cipher to decrypt the message. Spaces and other punctuation in the message should be left as-is, and do not advance your positions in the key sets.

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 three lines:

- The encrypted message, which contains uppercase letters, spaces, and punctuation, and may be of any length.
- A list of at least two positive integers between 1 and 25 inclusive, separated by spaces, representing the shift values that were used to encrypt the message.
- A list of at least two integers, each either 0 or 1 and separated by spaces, representing the direction values that were used to encrypt the message.

```
3
EDEPCG
1 1 2
1 0 1
EDENEC
1 1 2
1 0
DYA! EZP?
1 2
1 0 0
```

Sample Output

For each test case, your program must print the original, plaintext message. Print the message on a single line in lowercase letters, preserving any spaces or punctuation.

```
decode
decode
cab! car?
```

Problem 15: Countdown to Launch

Points: 40

Author: Ben Fenton, Ampthill, Reddings Wood, United Kingdom

Problem Background

In July 2019, Lockheed Martin began work with the UK Space Agency to construct a new satellite launch complex in Thurso, Scotland. The United Kingdom has been a leader in satellite production for years, but has had to rely on other countries to actually get them into space. With this new launch facility, set to open early this decade, the UK will be able to cement its position at the forefront of spacefaring countries.

Even if you haven't been to Scotland, you're probably aware that it's famous for a number of things: bagpipes, haggis, and its almost constantly wet climate. While the opening ceremonies for the launch site will likely include a large number of the first two items, the last one could prevent the UK's inaugural rocket launch from happening. Launching rockets is a difficult and precise operation, and poor weather can easily put a damper on launch plans.

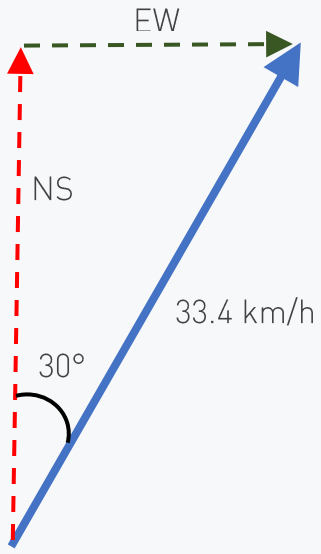
Problem Description

In order for a rocket launch to be carried out, a large number of conditions need to be met in order to ensure the success of the mission and the safety of the rocket and everyone in its vicinity. A launch must take place during a certain timeframe, known as the "launch window." Additionally, poor weather can prevent a launch from taking place, due to the risk that a rocket may be blown off target or become damaged during the launch. Launch times must be scheduled with all of these factors in mind.

As part of Lockheed Martin's efforts to build the new launch facility, your team is working on developing a system to recommend the ideal launch time for future missions based on their launch window and upcoming weather forecasts. The UK's rocket scientists recommend that a launch only take place if all of the following conditions are true:

- The cloud thickness is no higher than 1000 meters (1 kilometer)
- The windspeed along the North-South axis is no higher than 20 kilometers per hour (km/h)
- The windspeed along the East-West axis is no higher than 40 kilometers per hour (km/h)

The wind conditions will be reported as a speed and direction; you will need to use trigonometry to determine what the windspeed is across the respective axes. For example, if the wind is travelling at 33.4 km/h at a direction of 30°:



$$\sin\theta = \frac{wind_{EW}}{wind_{total}}$$

$$\cos\theta = \frac{wind_{NS}}{wind_{total}}$$

$$wind_{EW} = 33.4 * \sin 30° = 16.7$$

$$wind_{NS} = 33.4 * \cos 30° = 28.9$$

In this case, the windspeed across the East/West axis is within tolerances, but the windspeed across the North/South axis is too high. The launch will have to be delayed. Refer to the reference materials at the beginning of the problem packet for more information about the math demonstrated above.

Your system will automatically download the weather forecasts for 00:00, 06:00, 12:00, and 18:00 on each day within a mission's launch window. Your team must develop a program that is able to use that information to determine the earliest timeframe within the launch window that meets all of these conditions. In the event the weather is particularly bad and no timeframe within the launch window meets the conditions above, the program should recommend the launch be cancelled entirely.

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 a positive integer, X, representing the number of possible launch times within the launch window
- X lines containing information about the possible launch times and forecasted weather conditions at those times. Launch times will be presented in increasing chronological order. Values listed below are separated by spaces.
    - The date of the potential launch time, in YYYY-MM-DD format
    - The time of the potential launch time, in 24-hour HH:MM format
    - A non-negative integer value representing the expected cloud thickness in meters
    - A non-negative decimal value representing the expected windspeed at that time in km/h
    - An integer between 0 and 359 inclusive representing the expected wind direction at that time in degrees (0° = North, 90° = East, 180° = South, 270° = West)

```
2
4
2020-05-02 00:00 1100 25.0 45
2020-05-02 06:00 950 33.4 30
2020-05-02 12:00 875 22.2 60
2020-05-02 18:00 600 18.2 75
4
2020-05-03 00:00 800 22.0 180
2020-05-03 06:00 975 27.0 195
2020-05-03 12:00 1150 24.2 210
2020-05-03 18:00 1075 23.4 210
```

Sample Output

For each test case, your program must print the date and time, in 24-hour YYYY-MM-DD HH:MM format, of the earliest potential launch time meeting the safety parameters outlined above. In the event no launch time provided meets parameters, print the text "ABORT LAUNCH" instead.

```
2020-05-02 12:00
ABORT LAUNCH
```

Problem 16: Playing with Polynomials

Points: 45

Author: Kelly Reust, Denver, Colorado, United States

Problem Background

If you've taken an algebra course, you're probably familiar with polynomial expressions. These expressions usually take a form similar to the one shown below:

$$f(x) = a + bx + cx^2 + dx^3 + \cdots$$

The letters in this expression (except for x) represent the coefficients of each term in the equation. The small numbers appearing above most of the x's are the exponents of each term. Large equations like these are often used in physics; for example, the equation below can be used to find the distance travelled by an accelerating object.

$$d(t) = x + vt + 0.5at^2$$

Occasionally it's necessary to multiply polynomial expressions together to create an even larger expression. Multiplying small polynomials can be done by hand without too much trouble, but very large ones can prove difficult. Let's write a program to do the work for us.

Problem Description

When multiplying polynomial expressions, each term in the first expression needs to be individually multiplied with each term in the second expression. You might have learned the "FOIL" method in school for multiplying two-term polynomials together. To demonstrate, let's multiply the expressions $(2x + 3x^2)$ and $(4x + x^2)$.

The letters in FOIL stand for First, Outer, Inner, and Last, indicating how the terms should be multiplied together. We start with the first term in each expression:

$$2x * 4x = 8x^2$$

Just as in normal multiplication, we multiply the coefficients together. The exponents over the x's (which are both 1, since they're not displayed) get added together, however; remember that $x \times x = x^2$.

We repeat this process with the three other pairs of terms; the outer terms (first term of the first expression and second term of the second expression), the inner terms (second term of the first and first term of the second), and the last terms (of both expressions):

$$2x * x^2 = 2x^3$$
$$3x^2 * 4x = 12x^3$$
$$3x^2 * x^2 = 3x^4$$

Each of these new terms gets added together to form a new polynomial expression with four terms:

$$(2x + 3x^2) * (4x + x^2) = 8x^2 + 2x^3 + 12x^3 + 3x^4$$

We're not quite done yet, however. As you can see, there are two terms with an exponent of 3; $2x^3$ and $12x^3$. Since these terms have the same exponent, we can simplify the expression by simply adding their coefficients together. This results in a final answer of:

$$(2x + 3x^2) * (4x + x^2) = 8x^2 + 14x^3 + 3x^4$$

The FOIL method specifically applies to two-term polynomial expressions, but the same principle applies to polynomials with more terms. Each term in the first expression gets multiplied by each term in the second expression; multiplying coefficients and adding exponents. Once all the terms have been multiplied, simplify the expression by adding together coefficients that share an exponent.

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 representing polynomial expressions. Each line will contain between 2 and 9 (inclusive) integers separated by spaces, representing the coefficients of a polynomial expression. The first integer represents the coefficient paired with the exponent 0; the second, the coefficient paired with the exponent 1, and so on.

```
2
0 2 3
0 4 1
1 2 3 4
0 5 6
```

Sample Output

For each test case, your program must print a single line containing the polynomial expression obtained by multiplying the two given expressions together. Each term in the polynomial with a non-zero coefficient should be printed in order of increasing exponent. These terms should be separated from other terms by a plus sign (+) or minus sign (-) (use a minus sign when the following coefficient is negative), and should contain the following:

- The integer coefficient. The coefficient should not be printed if its absolute value is equal to 1 AND the exponent is not equal to 0.
- The lowercase letter x, a caret (^), and the integer exponent. The caret and exponent should not be printed if the exponent is less than or equal to 1. The letter x should not be printed if the exponent is equal to 0.

```
8x^2+14x^3+3x^4
5x+16x^2+27x^3+38x^4+24x^5
```

Problem 17: Flip Flop

Points: 45

Author: Richard Green, Whiteley, Hampshire, United Kingdom

Problem Background

Your coworker is… not great at his job. No matter how simple a task he's given, he always manages to make a mistake somehow. When asked to code something in Python, he went to the zoo. When asked to write a program in Java, he brewed coffee. When asked to sort something alphabetically, he sorted it numerically… which in hindsight was something of an achievement, since the list didn't contain any numbers.

Normally you'd simply try your best to avoid him, but whenever your coworker messes something up, it falls to somebody else to fix the problem. Unfortunately, you're so good at your own job that "somebody else" often ends up being you, since you're the only one with time to spare.

Problem Description

Today your confounding coworker was asked to put a bunch of data into a table. Fortunately, he didn't carve the numbers into the conference table this time, but it's still not correct. All of the data is transposed from where it should be. Each row in the table should actually be a column, and vice versa. Since you've already written a program to read in the data in the intended format, it'll all have to be flipped around.

$$\begin{bmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

As shown above, each table with X rows and Y columns needs to be flipped so it has Y rows and X columns. Every data point's coordinates in the table need to be reversed as well; the '2' in row 2, column 1 needs to be moved so that it's in row 1, column 2.

You've already wasted enough time fixing your coworker's mistakes, and there's a lot of data to fix. Maybe you could write a program to do most of the work for you?

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 two positive integers, separated by spaces:
  - R, representing the number of rows in your coworker's table

    - C, representing the number of columns in your coworker's table
- R lines, each containing up to C table cells with integer values, separated by commas (,). Some cells in the table may be empty.

```
2
5 2
1,11
2,12
3,13
4,14
5,15
4 6
1,2,3,,5,6
11,12,13,14,15,16
21,,23,24,,26
,32,,,35,
```

Sample Output

For each test case, your program must print the transposed (correct) version of your coworker's table. Within each row, separate cells with commas (,) and keep empty cells empty.

```
1,2,3,4,5
11,12,13,14,15
1,11,21,
2,12,,32
3,13,23,
,14,24,
5,15,,35
6,16,26,
```

Problem 18: Assemble the Team

Points: 50

Author: Brett Reynolds, Annapolis Junction, Maryland, United States

Problem Background

The People in Dark Suits have received word that aliens are planning to invade the Earth, and need to put together a team to defend the planet. They need as much help as they can get, but they also need to make sure the team can work together effectively, or else the human race is doomed! Fortunately, using advanced alien technology, the People in Dark Suits are able to perfectly represent their agent's personalities with a numeric score. Agents with similar scores are more likely to get along, but agents with very different scores could end up fighting each other instead of the invaders.

Your task is to put together as large a team as you possibly can while making sure the team can remain focused on the task at hand, and not be distracted by personal squabbles. The fate of the Earth depends on you!

Problem Description

Each agent of the People in Dark Suits is represented by a single uppercase letter. The personnel records you have access to associate those letters with a numeric score between 1 and 100, inclusive. This is the agent's personality score. As mentioned above, two agents with a large gap between their personality scores are unlikely to work well together. Due to the serious nature of this mission, you need to ensure that the maximum difference in personality scores between any two members of your team is no greater than 10. At the same time, you need to select as many team members as possible that fit within this range. Once your team is selected, you'll need to notify the agents of their assignments.

In the event you are able to form two or more teams of equal size that meet the personality requirements, select the team with the agents that have the letters appearing earlier in the alphabet. These agents have seniority and are more likely to be successful against the invaders. For example, if one team contains Agent A, and another contains Agent B, pick the team with Agent A. If Agent A is in both teams, break the tie with the next agents alphabetically, and so on.

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 containing information about the agents, separated by spaces. Each agent's information will be presented as follows:

- An uppercase letter, representing the agent's designation
- An equals sign (=)
- A positive integer between 1 and 100 inclusive, representing the agent's personality score

```
2
A=10 B=22 C=15 D=18 E=4
F=66 G=52 H=54 I=56 J=58 K=60 L=62 M=64
```

Sample Output

For each test case, your program must print the letters of the agents selected for the team on a single line, in alphabetical order, separated by spaces.

```
A C D
F I J K L M
```

Problem 19: Reschedule It!

Points: 55

Author: Wojciech Kozioł, Mielec, Poland

Problem Background

The Lockheed Martin F-35 Lightning II is a versatile stealth fighter currently in use by air forces around the world. With three primary variants to allow the fighter jet to be deployed from land or aircraft carriers, the F-35 stands to be the dominant force in the air for decades to come.

Lockheed Martin manufactures the F-35 in its production facility outside of Fort Worth, Texas. With high demand for the aircraft from multiple branches of the US military, not to mention the armed forces of other countries, it's important that the production lines move smoothly. At the same time, Lockheed Martin can't afford to have a bunch of extra aircraft just sitting around; the cost to store, secure, and maintain them can quickly add up over time. A great deal of planning and management is needed to ensure that production proceeds at exactly the right pace.

Problem Description

You're working in production management at Lockheed Martin Aeronautics on the F-35 manufacturing line. Your task is to make sure that production is proceeding at the correct pace, or if changes need to be made to the schedule. You are working primarily with two sets of data: the production schedule, which tells you how many aircraft will come off the production line on which days; and purchase orders, which tell you how many aircraft need to be delivered on which days.

You need to ensure that you have enough aircraft on hand to fulfill every purchase order, but that you aren't producing extra aircraft that nobody wants. There are also two more restrictions that you have to keep track of:

- Once an aircraft comes off the production line, it cannot be sold until at least the next day. This allows time for US Air Force pilots to conduct test flights on the new aircraft.
- Aircraft must be sold within four weeks (28 days) after they come off the production line. Beyond this point, the aircraft would be due for routine maintenance, which would cut into Lockheed Martin's profits.

If at any point you do not have enough aircraft to fulfill a purchase order, or an aircraft is left in a hangar for more than 4 weeks, your production schedule has failed. Your team must write a program to analyze the production and order schedules and highlight these problems before they have a chance to occur.

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 two positive integers, separated by spaces:
  - P, representing the number of entries in the production schedule
  - O, representing the number of purchase orders
- P lines containing information about the production schedule. Production schedule information is presented in chronological order. Each line will contain the following information, separated by spaces:
  - A date, in YYYY-MM-DD format, showing the date on which one or more aircraft will be completed
  - A positive integer representing the number of aircraft to be completed on that date
- O lines containing information about purchase orders. Purchase orders are presented in chronological order. Each line will contain the following information, separated by spaces:
  - A date, in YYYY-MM-DD format, showing the date on which one or more aircraft must be delivered
  - A positive integer representing the number of aircraft to be delivered on that date

```
3
2 3
2020-06-15 5
2020-06-25 4
2020-06-16 3
2020-06-26 3
2020-07-02 3
3 2
2020-06-15 2
2020-06-24 2
2020-06-25 2
2020-06-26 3
2020-07-25 3
2 2
2020-06-15 2
2020-06-24 3
2020-06-26 2
2020-06-25 2
```

Sample Output

For each test case, your program must determine if the given production and order schedule has problems. If the schedule is free of issues, print "OK". If the schedule has one or more problems, print "NOT OK".

```
OK
NOT OK
NOT OK
```

Problem 20: Bring John Glenn Home

Points: 60

Author: Wesley Holcombe, Colorado Springs, Colorado, United States

Problem Background

The movie "Hidden Figures" focused on the work of Katherine Johnson, an African-American woman working at NASA during the Mercury missions. At that time in history, computers were still uncommon, and so Johnson worked as a "human computer," performing complex calculations to ensure that missions were executed successfully. Despite her crucial contributions to the program, Johnson faced a great deal of discrimination from her colleagues. Only recently have Johnson and her fellow African-American colleagues received due recognition for their work in space exploration; she was awarded the Presidential Medal of Freedom in 2015, and the Congressional Gold Medal in 2019. Katherine Johnson died in 2020 at the age of 101.

In one key scene of the movie, Johnson described how to calculate the trajectory of John Glenn's space capsule using "Euler's Method."

Problem Description

Euler's Method is an ancient approximation technique used for differential equations, commonly referred to as "initial value problems." To use it, you solve the equation for the given initial values, then step to the next value based on the slope of the line calculated.

Here, we'll be attempting to approximate the solution for the formula:

$$y = f(x) = \int \frac{\sin(x)}{x} dx$$

If you haven't studied calculus yet, don't worry; we're only going to approximate the solution to this function, not solve it in full. The only bit of calculus we really need is to determine the "derivative" of this function, which is:

$$f'(x) = \frac{\sin(x)}{x}$$

The formula we'll actually be solving for each iteration of Euler's Method is this:

$$y_n = y_{n-1} + h * f'(x_{n-1})$$

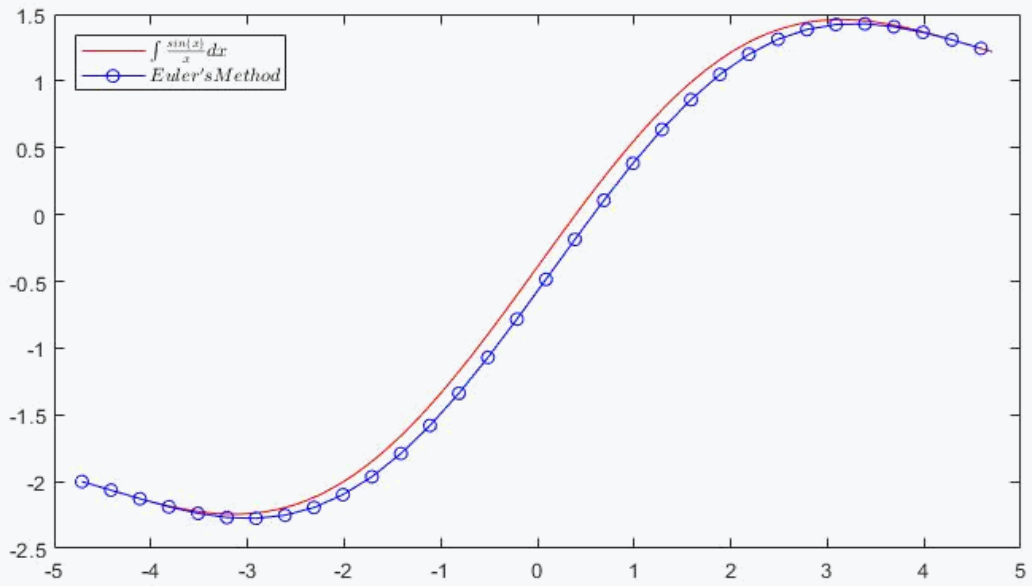
In this formula, x and y represent coordinates on a grid, and h is the step size used for the approximation; the amount by which x increases during each step. To use Euler's Method, we

have to start with a set of initial values, x_0 and y_0. Whenever the value for x is 0, assume that $f'(0) = 1$.

For example, let's try to approximate this equation using the initial values of:
$x_0 = -1.5\pi, y_0 = -2, h = .3$.

This table and chart show how the values of x and y change following each step of the equation, and how they compare to the actual solution to this equation.

n	$y_n = y_{n-1} + h * \left(\frac{\sin x_{n-1}}{x_{n-1}}\right)$	$x_n = h + x_{n-1}$
0	-2	-1.5π = -4.7124
1	-2.0637	-4.4124
2	-2.1286	-4.1124
3	-2.1888	-3.8124
4	-2.2377	-3.5124
5	-2.2687	-3.2124
6	-2.2753	-2.9124



For this problem, you will need to use Euler's Method to provide several approximated values for the formula given above.

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 the following values, separated by spaces:

- x_0, a number representing the initial x value.

- y_0, a number representing the initial y value.
- h, a number representing the constant step value.
- n, a positive integer representing the total number of iterations to perform.

```
2
1 5 0.5 6
-.54 0 0.01 8
```

Sample Output

For each test case, your program must print a single line containing the value of y_n, the value of y obtained after performing n iterations. Values should be rounded to 3 decimal places, not including any trailing zeroes.

```
6.074
0.077
```

Problem 21: Around the Town

Points: 60

Author: Anthony Vardaro, Dallas, Texas, United States

Problem Background

Public transportation systems often consist of a variety of transportation methods designed to make it easier for people or groups to move around a city. Whatever the method of travel, any public transportation system will have a finite number of "stops," where patrons gather to wait for the arrival of a bus or train. It's crucial that patrons should not have to walk too far to reach a stop, and so stops are often strategically placed near tourist attractions, residential areas, and other high population areas where a stop would be useful.

Lockheed Martin has been contracted to help a city design their first public transportation system. They are able to provide your team with the locations of several major landmarks around the city, but need your help making sure that their proposed bus stops are in the best possible locations.

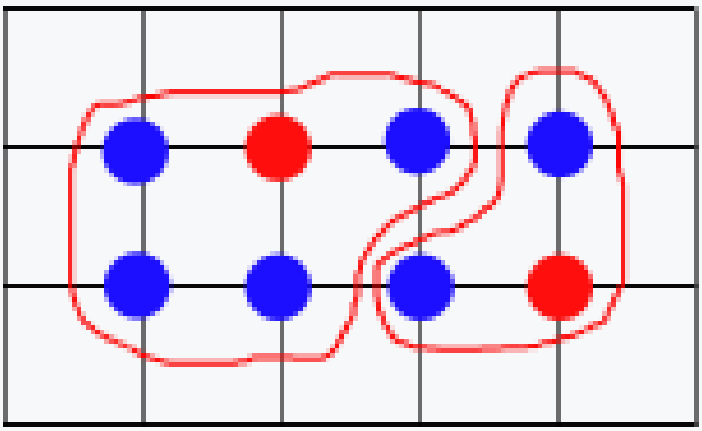
Problem Description

The city council has a rough idea of where they'd like bus stops to be located, but are concerned that they may not be in the best possible location. Your lead engineer suggests using a type of heuristic algorithm called "k-means" to optimize the stop locations.

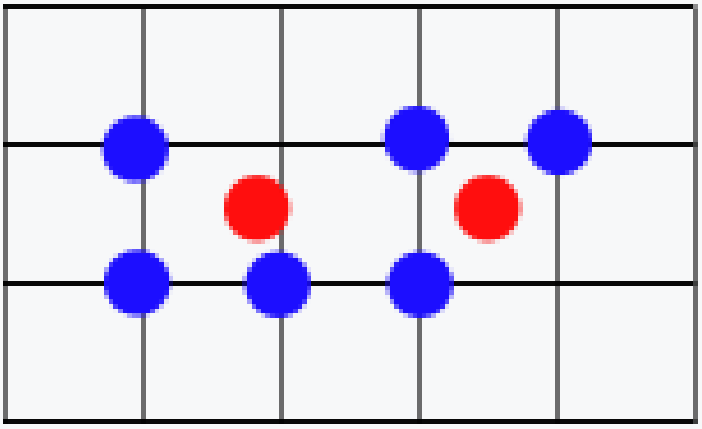
The k-means algorithm works by collecting a series of points into clusters. Each cluster has a "centroid;" a point which is the minimum possible distance from each data point within its cluster. Normally, the algorithm starts by choosing a set of centroids at random, then works to optimize them; in this case, we'll start with the city council's proposed bus stops as our starting set of centroids.

The algorithm works over several iterations; in each iteration, points (landmarks) are sorted into clusters by identifying which candidate centroid (bus stop) is closest to them. (In the event of a tie, landmarks should prefer the bus stop with the lowest-value X coordinate; if still tied, the bus stop with the lowest-value Y coordinate.) Each cluster then identifies a new centroid by averaging the coordinates of all points within the cluster. The clusters are then disbanded and the process repeats, stopping when none of the centroids change position following an iteration.

For example, let's look at the coordinate grid below. The blue points represent landmarks, and the red points represent the city council's proposed bus stops.



The wavy red lines indicate which landmarks have been grouped into each cluster; again, this is done by identifying the closest candidate bus stop to each landmark. With the clusters formed, we can average the coordinates of each landmark within the cluster to identify the new bus stop locations. The left-hand cluster's bus stop would move down and slightly to the left as a result; the right-hand cluster's bus stop would move directly between its two landmarks, as shown below.



The process then repeats, forming new clusters and moving the bus stops further. You'll notice that the top center landmark is now closer to the right-hand bus stop than it is the left-hand one; it will switch to the right-hand cluster as a result. These new clusters are used to identify new bus stop locations, until we complete an iteration without moving the locations at all.

The following formula for determining the straight-line distance between two points will be helpful as you solve this problem:

$$d = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

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 two positive integers, separated by spaces:

    - L, the number of landmarks requiring access to public transportation
    - S, the number of candidate bus stops identified by the city council
- L lines containing two integers, representing the X and Y coordinates (respectively) of a landmark in the city
- S lines containing two integers, representing the X and Y coordinates (respectively) of a candidate bus stop

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

Sample Output

For each test case, your program must print the X and Y coordinates of the optimum location of each bus stop. Round coordinates to one decimal place, include any trailing zeroes, and separate them by spaces; each bus stop should be printed on a separate line, in the order provided.

```
0.3 0.3
2.3 0.7
```

Problem 22: Dive Time

Points: 65

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

Problem Background

SCUBA diving can be an amazing experience, but divers have to overcome one major obstacle: breathing. Divers typically breathe from a tank of compressed air using specialized equipment that allows them to breathe almost normally. However, the body is subject to very high pressures when underwater, which can cause some difficulties of its own.

Most of the air we breathe is comprised of nitrogen, which our bodies largely ignore. However, when in a high-pressure environment (such as when someone is diving), the nitrogen is forcibly dissolved into our blood. As a diver returns to normal pressure, the nitrogen precipitates, or begins to return to a gaseous form. As long as this return is done gradually, the nitrogen leaves via the lungs, gets breathed out, and there aren't any problems.

However, if a diver ascends too quickly, the nitrogen remains in the bloodstream as it precipitates, forming small bubbles. This causes an extremely painful and potentially fatal condition called decompression sickness, more commonly known as the "bends" for how it causes victims to double up in pain. To avoid this condition, divers must ascend very slowly, and in many cases must wait at certain depths for a period of time to allow the nitrogen enough time to leave the bloodstream. These "decompression stops" are determined using a complex series of calculations... which, in case you were wondering, is finally where software engineering gets involved.

Divers can make use of computers that measure how long a diver remains at a particular depth, and automatically determine at what depths a diver has to make a decompression stop, and for how long. Since performing these decompression stops accurately can mean the difference between life and death, it's critical that these computers perform accurate calculations.

Problem Description

Lockheed Martin has been contracted by the United States Navy to develop a new system to calculate required decompression stops for SCUBA divers. The US Navy maintains a set of commonly-used dive tables which outline the required decompression stops for dives of various durations and depths. Your program will need to use the information in these tables, which will be provided to your program, to determine the appropriate decompression schedule.

Dive tables can vary widely, depending on a wide range of factors. Your program will be provided with dive table information appropriate to the circumstances of the dive being undertaken.

To use a dive table, a diver must determine the maximum depth they reached, then find the entries in the table that show a depth equal to or greater than that depth. Then, they must calculate their "bottom time;" the amount of time spent between entering the water and starting their ascent. They must find any entries in the table that pertain to the next time greater than or equal to their bottom time. These entries will indicate how long they need to stop for decompression, and at what depth(s).

For example, consider a diver that reached a depth of 115 feet and started returning to the surface after spending 38 minutes underwater. The table below shows an excerpt of the dive table they are using.

Max Depth (feet)	Stop Depth (feet)	Decompression Schedules for Bottom Times (minutes)								
110		20	25	30	35	40	45	50		
	20		5	14	27	39	50	71		
120		15	20	25	30	35	40	45		
	30						2	3		
	20		4	9	24	38	49	71		
130		12	15	20	25	30	35	40		
	30					2	5	6		
	20		3	8	17	32	44	66		

The diver exceeded a depth of 110 feet, so cannot use the information in that section of the table. They must use the next-deeper section, for dives up to 120 feet. The diver didn't start ascending until after 38 minutes, so they must use the schedule for the first time greater than or equal to that amount; in this case, the 40-minute schedule. That schedule indicates they must make two decompression stops; one for two minutes at 30 feet, and another for 49 minutes at 20 feet.

Your program must read in provided dive table information and data about individual dives to determine the appropriate decompression schedule to use for each dive. As shown above, some dives may require multiple decompression stops; in this case, you will receive multiple dive table entries for the same depth and bottom time. Some very short or very shallow dives may not require any decompression stops; these will be represented with dive table entries that indicate a decompression stop of 0 minutes is required at a depth of 0 feet.

Safety Disclaimer: The information given here, and in any inputs provided to your program, is provided as a reference for solving this problem only. It has been simplified from the real US Navy dive tables, but, as a result, is incomplete. DO NOT USE THIS INFORMATION FOR ACTUAL SCUBA DIVES. Contact your diving certification authority, consult with your divemaster, and/or use a real dive computer (not one you wrote yourself!) to determine the appropriate decompression schedule for any SCUBA dives you make. As stated above, SCUBA diving is a dangerous sport that requires extensive training and adherence to strict safety protocols to avoid permanent bodily harm or death.

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 two positive integers, separated by a space:
  - X, the number of entries in the dive table to be used for the test case
  - D, the number of dives conducted using that dive table
- X lines, each representing an entry in the dive table, and containing four integers separated by spaces, each representing:
  - The maximum depth (in feet) for this entry. This value will be positive.
  - The maximum bottom time (in feet) for this entry. This value will be positive.
  - The depth (in feet) at which a compression stop is required. This value will not be negative; a value of 0 indicates no compression stops are required.
  - The time (in minutes) at which a diver must stop at that depth for decompression. This value will not be negative; a value of 0 indicates no compression stops are required.
- D lines, each representing a dive performed using the given dive table, and containing two positive integers separated by spaces, each representing:
  - The maximum depth (in feet) reached by the diver. This value will not exceed the maximum depth defined by the dive table.
  - The diver's bottom time (in minutes). This value will not exceed the maximum bottom time for the given depth, as defined by the dive table.

```
1
5 3
20 600 0 0
30 371 0 0
30 380 20 5
120 40 30 2
120 40 20 49
28 375
16 240
117 38
```

Sample Output

For each test case, your program must output information about the decompression stops the divers must make during their ascents, if any, in order to avoid decompression sickness.

For each stop a diver is required to make in each dive, your program must print one line containing two integers separated by spaces, each representing:

- The depth (in feet) at which a decompression stop should be made
- The amount of time (in minutes) at which the diver should remain at that depth

When multiple decompression stops are required, they must be printed in order of decreasing depth. If a dive does not require any decompression stops, your program should print the words "No Stop".

```
20 5
No Stop
30 2
20 49
```

Problem 23: The Last Place You Look

Points: 65

Author: Ben Fenton, Ampthill, Reddings Wood, United Kingdom

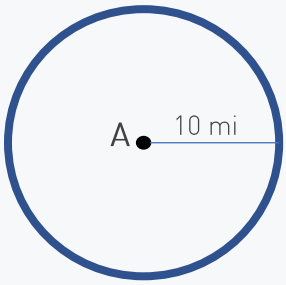
Problem Background

It's said that when you're looking for something, you always find it in the last place you look. This is invariably true, because once you find the item, you stop looking for it, but more often than not it seems that the item also happens to be in the last place you'd ever think of looking. Let's reduce the guesswork somewhat and create some devices that can guarantee that whenever our items walk off on their own, they're always in the first place we look!

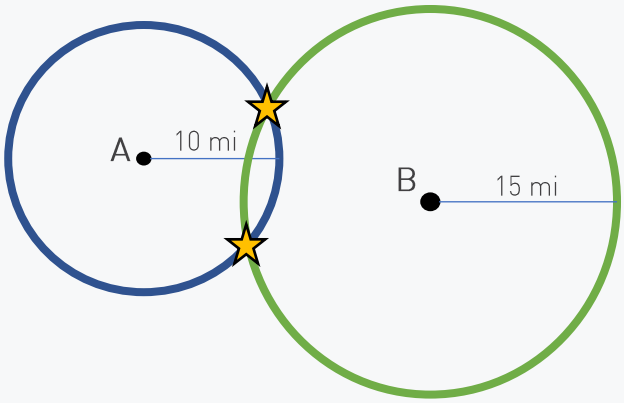
Geopositioning tags were among the first products to make up the rapidly-growing "Internet of Things;" an industry of normally common-place items that have a wide range of useful features added to them by virtue of a constant wireless internet connection. In this case, a keychain could have a small fob connected to it that contains a miniaturized GPS receiver. This fob constantly reports its location to an app on your smartphone, allowing you to easily find your keys if they should get lost. The concept may sound simple, but a lot of software design goes into something like this.

GPS, or Global Positioning Systems, rely on a network of satellites orbiting far above the earth. Each satellite is able to track its position over the Earth, and constantly broadcasts its location and the current time. A GPS receiver can use this information to determine how far away it is from the satellite's position. However, a signal from a single satellite is not enough to determine location; in actuality, a receiver needs signals from at least four different satellites to pinpoint its own location.

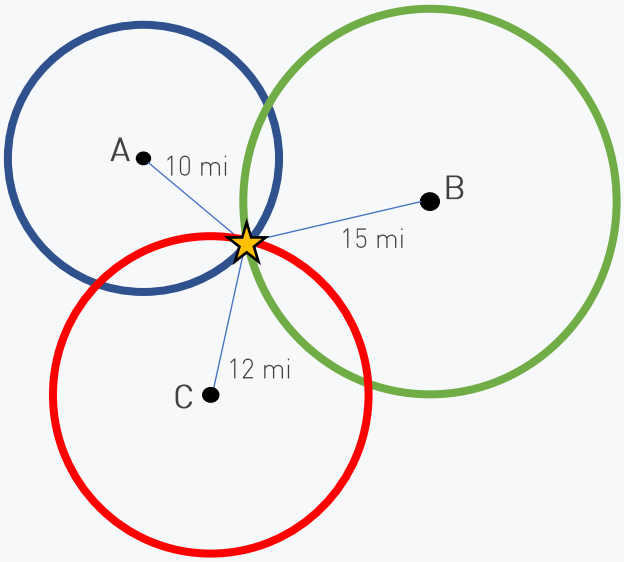
Let's look at a simplified example of how GPS receivers work. If a receiver only knows it is 10 miles away from point A, it could be located anywhere along the circumference of this circle:



Even a signal from a second satellite can't narrow down an exact position; the receiver could be in either of the two locations where these circles intersect:



Once we add a third signal, we can get a better idea of our location. The receiver must be located at the one point at which it can be the indicated distance from each of the three satellites:



Again, in practice, a receiver would actually need four satellite signals to confirm its location; since GPS satellites work in three dimensions (latitude, longitude, and altitude), you need four readings to get an exact location. For this problem, we'll ignore the altitude, allowing us to focus on the first two dimensions.

Problem Description

Lockheed Martin is working on a set of devices that can work as a sort of local GPS to help aircraft maintainers find tools they've misplaced; a set of three transmitters that act as "GPS satellites," and small receiver tags that attach to tools. Since using latitude and longitude over an area as small as an aircraft hangar isn't going to be terribly accurate, your team is developing an alternative coordinate system for use with these tags.

The hangar will be divided into a grid using an integer-based X,Y coordinate system. Receiver tags will be able to determine the "taxicab distance" between them and each of the three transmitters spaced around the hangar. "Taxicab distance" is how many spaces in the

coordinate grid must be traversed to reach your destination while moving only vertically and horizontally, as though you were a taxi navigating city streets. The grid below shows the "taxicab distance" between each cell and the target cell, marked with an X.

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

As you can see, each number creates a diamond pattern around the central X, approximating the circles shown in the GPS diagrams above.

Using the known locations of the three transmitters, and the "taxicab distance" to each of them, you must determine the location of the receiver tag (and the tool to which it is attached).

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 integer values, separated by spaces:

- The X-coordinate of the first transmitter. This will be an integer value.
- The Y-coordinate of the first transmitter. This will be an integer value.
- The taxicab distance of the receiver from the first transmitter. This will be a positive integer value.
- The X-coordinate of the second transmitter. This will be an integer value.
- The Y-coordinate of the second transmitter. This will be an integer value.
- The taxicab distance of the receiver from the second transmitter. This will be a positive integer value.
- The X-coordinate of the third transmitter. This will be an integer value.
- The Y-coordinate of the third transmitter. This will be an integer value.
- The taxicab distance of the receiver from the third transmitter. This will be a positive integer value.

```
3
10 10 20 -5 -5 10 8 -8 16
8 8 15 -10 5 10 6 -2 13
-4 5 6 9 4 10 5 -4 14
```

Sample Output

For each test case, your program must print a single line containing the integer X- and Y-coordinates of the receiver tag, in (X,Y) format.

```
(0,0)
(-2,3)
(1,6)
```

Problem 24: Three's Company

Points: 70

Author: Louis Ronat, Denver, Colorado, United States

Problem Background

Computer programs are complicated. However, analyzing just *how* complicated computer programs are - determining their computational complexity - is a fundamental topic in computer science. Different classes of computational complexity exist, and one of the biggest unproven questions in complexity analysis deals with the class known as NP problems. An NP problem is one where it is believed that an algorithm solving the problem must take an amount of time that grows faster than a polynomial compared to the size of the input. That is, if a is a constant and n is the size of the input, the time needed to solve the problem increases faster than the value of n^a. Many of the most difficult Lockheed Martin Code Quest problems fall into this category.

As complicated as these problems are to solve, verifying that a particular solution is correct is not as difficult. NP problems have a property where verifying a provided solution can be completed in polynomial time. Today, you'll be verifying one such NP problem, known as 3-satisfiability or 3-SAT.

Problem Description

3-SAT problems start with a logical statement grouped in a specific manner. The statement includes two or more "triplets" of Boolean inputs, joined with a logical OR statement. One such triplet is shown below:

```
A or B or C
```

Each triplet is then joined together with a logical AND to form the full logical statement. For example:

```
(A or B or C) and (!D or !E or !F)
```

Note that the ! character appears here as a negation; !D is true when D is false.

Solving a 3-SAT problem requires answering a question: "Will this statement ever be true?" The statement given above will be true if at least one of the inputs A, B, or C is true, and at least one of the inputs D, E, and F is false. As long as some combination of possible inputs matches those criteria, the answer in this case is yes; if no such combination of inputs is valid, the answer is no.

For example, consider if that statement were placed in an if statement in your programming language. The 3-SAT problem's answer is "yes" if there is some way to execute the code inside that if block. If there is no way to access that code, the answer is "no."

A 3-SAT can have more than two triplets; in each case, the members within a triplet are joined using a logical OR, and each triplet is joined with a logical AND to form the complete statement.

Your team must create a program that can build a given 3-SAT statement and test it against several inputs to determine if the problem is solvable. Be careful; the true/false inputs will be provided in alphabetical order, so you'll need to be sure to correctly map them to the variables in the triplets.

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 two positive integers separated by spaces:
  - The first integer, T, represents the number of triplets in the 3-SAT statement. T will be greater than or equal to 2.
  - The second integer, I, represents the number of inputs to test against the final statement.
- T lines containing three input values, separated by spaces, representing the triplets that will form the 3-SAT statement. Each line represents a single triplet, and each input value will consist of a single uppercase letter. Letters may be prefaced with an exclamation point (!), indicating that that input should be negated when added to the 3-SAT statement. Within each test case, the letters used will start with A and will not skip any letters, however the letters may not be presented in alphabetical order.
- I lines, each containing 3T 0's or 1's separated by spaces, representing the input values to be tested against the 3-SAT statement. A value of 0 represents 'false', and a value of 1 represents 'true'. Values are presented in alphabetical order; the first value represents the value of A; the second, the value of B, and so on.

```
1
2 4
A B C
!D !E !F
1 0 0 1 1 0
1 1 1 1 1 1
0 0 0 0 0 0
1 1 1 0 0 0
```

Sample Output

For each test case, your program must output one line for each set of values to be tested, containing the word "TRUE" if the 3-SAT statement valuated to 'true' with the given values, or "FALSE" otherwise.

```
TRUE
FALSE
FALSE
TRUE
```

Problem 25: Playfair Cipher

Points: 75

Author: Brett Reynolds, Annapolis Junction, Maryland, United States

Problem Background

Substitution ciphers are methods of encrypting text that involve replacing each letter in the "plaintext" with another letter to generate a "ciphertext." While quick and easy to implement and remember, they are inherently weak ciphers. Certain letters appear more frequently than others, allowing a codebreaker to make educated guesses about what certain letters represent. By the mid-19th century, such methods were well known to government forces, and intercepted messages could be broken with relative ease.

In 1854, English inventor Sir Charles Wheatstone created a cipher intended to solve this problem. His cipher - named the Playfair cipher, after its supporter, Lord Playfair - replaced letters in pairs rather than each letter individually. Since the English language contains 26 letters, encrypting letters in pairs meant there were potentially over 600 possible pairings, making frequency analysis impractical. The Playfair cipher was also easy to use, and so it remained in use by military forces through World War II and the advent of the computer.

Problem Description

The Playfair cipher uses a word or short phrase as a keyword, which is used to build an encryption table. The encryption table is a five-by-five square filled with the letters of the English alphabet (for the purposes of this problem, the letter 'J' will be omitted; we'll cover how to handle it later).

For example, if the keyword is "PLAYFAIR DEMO", we would start by removing spaces and repeat letters from the keyword. In this case, the letter A appears twice, so its second occurrence is removed: "PLAYFIRDEMO." We then fill these letters into the five-by-five grid, starting in the top left corner and moving from left to right, top to bottom:

P	L	A	Y	F
I	R	D	E	M
O				

The remaining letters of the alphabet (again, except J) are then filled into the grid in order. Any letter already present in the table (in the keyword) is skipped.

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

This completes the encryption table, and it can now be used to encrypt (or decrypt) a message. To begin with, the letters in the original message are broken into pairs. Since we've omitted the letter 'J' from the encryption table, we'll also replace any instance of the letter 'j' with the letter 'x' during encryption. If we encrypt the phrase "code quest," this results in:

co de qu es tx

Notice that since "code quest" contains an odd number of letters, we've added an 'x' at the end to complete the final pair. Each pair of letters is then encrypted according to the following rules:

1. If both letters are the same (e.g. "ss"), replace the second letter with an 'x' ("sx") and continue encryption with the following rules.
2. If the letters appear in the same row of the encryption table, replace each letter with the one to its immediate right in the table (wrapping around to the left side as needed).
3. If the letters appear in the same column of the encryption table, replace each letter with the one immediately below it in the table (wrapping around to the top side as needed).
4. If the letters share neither a row nor column, replace each letter with the one on the same row as itself, but in the same column as its partner.

To demonstrate with our earlier example (keyword: "PLAYFAIR DEMO", plaintext "code quest"), these rules would be followed as shown:

Original Pair	Rule Applied	Encryption Table					Result Pair
co	2 - Same row; replace each letter with the one to its right	P	L	A	Y	F	GB
		I	R	D	E	M	
		O	B	C	G	H	
		K	N	Q	S	T	
		U	V	W	X	Z	
de	2 - Same row; replace each letter with the one to its right	P	L	A	Y	F	EM
		I	R	D	E	M	
		O	B	C	G	H	
		K	N	Q	S	T	
		U	V	W	X	Z	

Original Pair	Rule Applied	Encryption Table					Result Pair
qu	4 - Different row/column; replace each letter with the one in its row and its partner's column	P	L	A	Y	F	KW
		I	R	D	E	M	
		O	B	C	G	H	
		K	N	Q	S	T	
		U	V	W	X	Z	
es	3 - Same column; replace each letter with the one below it	P	L	A	Y	F	GX
		I	R	D	E	M	
		O	B	C	G	H	
		K	N	Q	S	T	
		U	V	W	X	Z	
tx	4 - Different row/column; replace each letter with the one in its row and its partner's column	P	L	A	Y	F	SZ
		I	R	D	E	M	
		O	B	C	G	H	
		K	N	Q	S	T	
		U	V	W	X	Z	

The resulting ciphertext, then, is "GB EM KW GX SZ." Note that the letter appears twice in our ciphertext, in place of the letters 'c' and 'e' - this is part of what foils attempts at frequency analysis. Depending on how the letters get paired up, a single letter in the ciphertext could represent any letter in the plaintext.

For this problem, you must write a program that decrypts the Playfair cipher, given the encrypted message and the keyword used to encrypt it. Decryption works in a similar manner to encryption; just make sure to move to the left or up when letters appear in the same row or column, respectively. Do not worry about replacing any 'x' letters that appear in the plaintext with 'j's or spaces; leave them as they appear.

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 the following information, separated by spaces:
  - A positive integer, **X**, representing the number of lines in the ciphertext
  - The keyword used to encrypt a message using the Playfair cipher, which will contain only uppercase letters.
- X lines containing the ciphertext to decrypt, consisting entirely of uppercase letters.

```
2
1 PLAYFAIRDEMO
GBEMKWGXSZ
2 LOCKHEEDMARTIN
KRLTUZBIDIBK
PLDIHGKH
```

Sample Output

For each test case, your program must print the plaintext obtained after decrypting the message, in lowercase letters.

```
codequestx
havefuntoday
goodluck
```

Problem 26: ASCII Squares

Points: 80

Author: Louis Ronat, Denver, Colorado, United States

Problem Background

Issac, a clever tile setter, has decided to charge his customers in a different way from his competitors in the hope of making some additional money off clients who don't notice a slight difference in the wording of his contracts.

Issac's competitors all charge based on the number of tiles laid down, but Issac plans to charge based on the number of *squares* laid down. He hopes most clients will think this is the same thing, since each tile is a square, but Issac also intends to charge them based on the larger squares formed by multiple tiles. In order to determine how much to charge, Issac has asked you to write a program to calculate the number of squares in a given tile layout. He's provided you with ASCII-art "sketches" of the tile layouts he's been hired to build.

Problem Description

Issac's tile layouts consist of pipes (|), underscores (_), and spaces. Your job is to count the number of "squares" formed in the layout.

A single tile, which constitutes a 1x1 square, looks like this:

```
 _
|_|
```

A 2x2 square must have a perimeter like the one shown below. The contents of the square may be empty (which occurs if there is a "hole" within the layout, surrounded by other tiles) or may contain some smaller squares.

```
 _ _
|   |
|_ _|
```

Similarly, for larger squares, an NxN square must have N underscores across the top and bottom, with each underscore separated by a space or a pipe. It must also have N pipes on both the left and right sides. Pipes and underscores may also appear within the shape.

Your program must count the number of squares - of any size - in the provided image. Note that non-square rectangles of any size are not counted.

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 a positive integer, X, representing the number of lines in the tile layout.
- X lines consisting of pipes, underscores, and spaces, representing the tile layout. Lines will not contain any trailing whitespace, and as such may not all be the same length, even within the same test case.

```
2
5
 _ _ _ _
|_|_|_|_|
|_|_|_|_|
|_|_|_|_|
|_|_|_|_|
4
 _ _ _ _
|_|_|_|_|_ _ _
  |_|_|_|_|_|_|
    |_|_|_|_|_|
```

Sample Output

For each test case, your program must print a single line containing an integer representing the total number of squares of any size present in the provided tile layout.

```
30
21
```

Problem 27: Shifty Bits

Points: 85

Author: Matt Marzin, King of Prussia, Pennsylvania, United States

Problem Background

Ground stations that command and control constellations of satellites must also be responsible for interpreting data from those satellites. To make data easier to transmit, data values from multiple sources are often combined into a single data string through a process called "commutation." Once received by a ground station, this data must be broken apart again so it can be properly analyzed; this process is called "decommutation" and the individual values are referred to as "measurands."

Problem Description

You're working with Lockheed Martin's Space Systems division to develop a new data decommutation algorithm. Your system will receive commutated data as a hexadecimal string; this must be translated into binary in order to properly separate the different measurands you'll be looking for. For example, the hexadecimal string 0xB312C675 would be converted like so:

Hex	B				3				1				2				C				6				7				5			
Decimal	11				3				1				2				12				6				7				5			
Binary	1	0	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	1	0	0	0	1	1	0	0	1	1	1	0	1	0	1

There are three main elements needed to identify and decommutate a measurand:

1. A data type, which determines how the information will be interpreted.
2. An offset marking where the data starts. The right-most bit will have an index of 0; the offset indicates at which index your data begins.
3. The length of the measurand, or the number of bits it contains.

The data types you'll encounter in this problem are listed below:

Data Type	Length	Minimum Value	Maximum Value
int	2 to 32 bits	-2,147,483,648	2,147,483,647
uint (unsigned int)	1 to 24 bits	0	16,777,215
float	32 bits	3.4E-38	3.4E38
double	64 bits	1.7E-308	1.7E308

Your programming language should have built-in functionality for converting binary values to float and decimal numbers; we strongly recommend using that. For int and uint values,

however, the bit length will vary, and may require your own interpretation. For int values, remember that regardless of the length of data, the leftmost bit will always indicate the sign of the value (positive = 0, negative = 1). Uint values will always be positive.

For example, let's use the data string presented above to extract an int measurand with a length of 6 and an offset of 4:

1	0	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	1	0	0	0	1	1	0	0	1	1	1	0	1	0	1

$$\left(-1 \times (1 \times 2^5)\right) + (0 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = ?$$
$$-32 + 0 + 0 + 4 + 2 + 1 = -25$$

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 a hexadecimal string, prefaced with '0x', and followed by any number of uppercase hexadecimal characters.
- A line containing a positive integer, **M**, representing the number of measurands contained in the data string.
- **M** lines, each containing the following information about one of the measurands to decommutate, separated by spaces:
  - A string indicating the data type of the measurand; one of int, uint, float, or double
  - A non-negative integer indicating the index offset at which the measurand begins
  - A positive integer indicating the length of the measurand in bits

```
2
0xDEADFACEBEEFBABE
3
double 0 64
int 8 16
uint 24 8
0x0123456789ABCDEF
2
int 5 6
float 16 32
```

Sample Output

For each test case, your program must print the value of each measurand in the order they were presented, one per line. Print doubles and floats to a precision of 5 decimal places; use lowercase scientific notation (as shown below) if the absolute value is greater than 99999.99999 or less than 0.00001. When scientific notation is required, print the exponent as **e+###** or **e-###** (as applicable), where ### is the three-digit exponent used, including any leading zeroes.

```
-1.19794e+148
-4166
190
-17
3704.60425
```

Problem 28: Labyrinth

Points: 90

Author: Brett Reynolds, Annapolis Junction, Maryland, United States

Problem Background

The term "labyrinth" has come to mean any large, difficult-to-navigate maze, but the word originates from Greek mythology. According to legend, the Greek Labyrinth was a massive, complex maze on the island of Crete, which held the fearsome half-human, half-bull Minotaur at bay. Anyone who entered the Labyrinth would become hopelessly lost and would eventually be devoured by its resident monster. According to that same legend, the hero Theseus entered the Labyrinth to slay the Minotaur and took a ball of thread with him to mark the path he followed so that he could eventually escape.

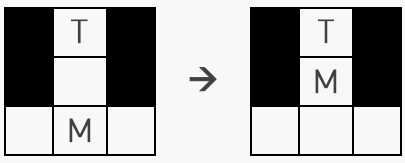
For this problem, we're going to change the legend somewhat. First, Theseus is entering the Labyrinth not to slay the Minotaur, but to steal a priceless treasure hidden inside. Secondly, Theseus is rather forgetful and has forgotten both his sword and his ball of thread; as a result, he can't defend himself against the Minotaur, and can't find his way back out once he gets the treasure. He'll need our help.

Problem Description

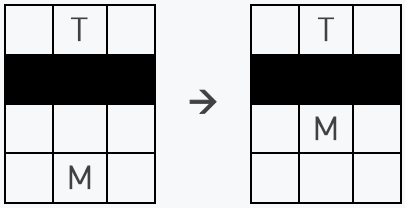
You will be provided with a map of the Labyrinth as viewed from above. On this map, you will be given the locations of both Theseus and the Minotaur. Theseus managed to find the treasure on his own without incident, but now he needs to escape as quickly as possible. You must find the shortest path Theseus can take to the exit that allows him to avoid the approaching Minotaur. If Theseus and the Minotaur ever run into each other (occupy the same position on the map), the Minotaur will eat Theseus.

Each cell within the map represents one step. Both Theseus and the Minotaur move at the same pace - one step at a time - but will take turns doing so. Theseus will move first, then the Minotaur, then Theseus again, and so on. Neither Theseus nor the Minotaur may move through walls or diagonally. The Minotaur may move through the exit as though it is an open space.

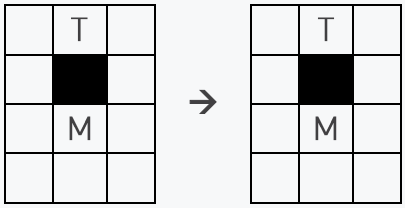
Fortunately for Theseus, the Minotaur isn't very smart, and will always move according to the following rules. In the diagrams below, "M" represents the Minotaur, "T" represents Theseus, and black cells are impassible walls.



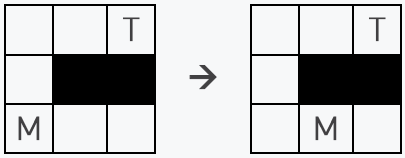
The Minotaur will always seek to reduce the straight-line distance between him and Theseus.



The Minotaur cannot move through walls, but will ignore walls when determining the distance between himself and Theseus.



If the only moves available to the Minotaur would increase the distance between himself and Theseus, the Minotaur will not move.



Given two choices that would result in an equal distance, the Minotaur will prefer to move left or right rather than up or down.

You program must calculate the fewest number of steps required to get Theseus safely to the exit. At each step, Theseus may move up, down, left or right. The Minotaur will then move in response to Theseus's new location. Theseus may also choose to remain still for one step; in this case, the Minotaur will still move on his turn, according to the rules above.

While the goal is to get Theseus to the exit as quickly as possible, be careful. If the Minotaur would ever move into the space occupied by Theseus, Theseus will be eaten and the path is invalid. Theseus may need to take a less direct route or lure the Minotaur into a dead end in order to safely reach the exit.

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 two positive integers separated by spaces, X and Y, representing the width and height of the map, respectively.
- Y lines, each containing X characters, representing the map of the labyrinth as follows:
    - An uppercase letter X represents an impassible wall.

- An uppercase letter E represents the exit from the labyrinth. There is only one exit in each map, and it may occur anywhere within the map.
- An uppercase letter T represents Theseus's starting position. This will occur only once in each map.
- An uppercase letter M represents the Minotaur's starting position. This will occur only once in each map.
- Spaces represent navigable hallways. The exit and both starting positions should also be considered navigable spaces.

```
1
8 7
XXXXXXEX
X      X
X XXXX X
X X M  X
X XXXX X
X     TX
XXXXXXXX
```

Sample Output

For each test case, your program must print a single line containing an integer representing the fewest number of steps required to allow Theseus to escape the maze unharmed.

```
15
```

Problem 29: Race to the Finish!

Points: 90

Author: Matthew Schmeiser, Montreal, Québec, Canada

Problem Background

We're putting you in the driver's seat! You need to drive your car around the track as quickly as possible, without crashing into any walls. Success will depend on skillfully controlling your car's speed and direction.

Problem Description

The racecourse will be represented by a grid, which will indicate your starting position and a 3x3 region representing the finish line. Your position and velocity at any point can be represented by a pair of numbers; for your position, your location along the X and Y axes (X,Y); and for your velocity, your current speed along each axis (V_x,V_y). At the beginning of the race, your velocity will be (0,0). Your velocity may never exceed an absolute value of 3 along either axis.

Your car will move along the racecourse in a series of steps. At each step, your car will move according to its current velocity. For example, if your velocity is (1,-2), your car will move one space upward and two spaces leftward. Following this movement, you will have the option to increase or decrease your car's speed by one along each axis. For example, you could change your car's velocity to (0,-2) (reducing your X speed), (2,-3) (increasing both your upward and leftward speed), or leave it at (1,-2) unchanged.

If at any point your car's position is outside the bounds of the provided racetrack, or your car's position is occupied by a wall, you will have crashed. If your car's position is within the 3x3 finish area, you have completed the course. Your goal is to determine the fewest number of steps required to bring your car from its starting position to a position within the finish area.

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 two positive integers separated by spaces, **Y** and **X**, representing the number of rows and columns in the racecourse, respectively.
- **Y** lines containing the layout of the racecourse. Each line will contain **X** characters, consisting of the following characters:
    - # (representing walls)
    - Spaces (representing open track)

- A single uppercase letter C (representing your starting position)
- Nine $ symbols in a 3x3 area (representing the finish area)

```
2
10 10
##########
# C ######
#   ######
#   ######
#   ######
#   ######
#     $$$#
#     $$$#
#     $$$#
##########
16 20
####################
########           #
#$$$               #
#$$$     #######   #
#$$$   #########   #
################    #
################    #
###############     #
##         ###     #
#                 ##
#    ######   ######
#    ###############
#   ################
#   ################
# C ################
####################
```

Sample Output

For each test case, your program must print an integer representing the least number of steps required to reach the finish area.

```
5
17
```

Problem 30: Checkmate

Points: 100

Author: Louis Ronat, Denver, Colorado, United States

Problem Background

Chess is an ancient game that still remains popular today. Since the early 2000's, however, the best chess players aren't human; they're algorithms programmed into computers. In fact, there are now multiple tournaments whose competitors are various "chess engines" - software programs designed to output valid chess moves in pursuit of a checkmate.

Problem Description

Your goal is to write an application that reads an input representing a chessboard and outputs whether the player set to move is in checkmate. The player to move can be deduced by determining which player is in check - if neither player is in check, no checkmate exists.

While chess is a well-known game, it rules are somewhat complicated, and so the ones relevant to this problem are reiterated below. If you are familiar with the game already, you can skip most of the rest of this description; however, make sure to take note of which letters are used to represent which pieces.

There are six different kinds of pieces in chess, all of which move in different ways. In this problem, they will be represented on the chessboard with upper or lowercase letters, as shown later in this section. The overall goal of the game is to force your opponent into a situation where their king is in danger, and they have no legal way to protect it. When a king is at risk of being captured, it's said to be "in check;" when no options exist to protect it, it's called "checkmate."

The six types of pieces and their rules of movement are as follows:

- P or p - Pawn
    - Can only move onto the space directly in front of it, with three relevant exceptions, explained below
    - A pawn cannot move forward if the space in front of it is occupied by another piece (of either color)
    - If an opposing piece is in either space diagonally ahead of a pawn, the pawn may move onto that space to capture that piece
    - A pawn in its starting position (second row from the player's side of the board) may move two spaces forward, if both spaces are unoccupied
- R or r - Rook

    - Can move any number of spaces horizontally or vertically
    - Cannot move through pieces of the same color
    - Can capture any piece of the opposing color by moving onto its space
- N or n - Knight
    - Moves two spaces either vertically or horizontally, then one space along the other axis (e.g. two up then one left, or two right then one down, etc.)
    - Can move through pieces of either color
    - Can capture any piece of the opposing color by moving onto its space
- B or b - Bishop
    - Can move any number of spaces diagonally
    - Cannot move through pieces of the same color
    - Can capture any piece of the opposing color by moving onto its space
- Q or q - Queen
    - Can move any number of spaces in any direction - horizontally, vertically, or diagonally
    - Cannot move through pieces of the same color
    - Can capture any piece of the opposing color by moving onto its space
- K or k - King
    - Can move one space in any direction - horizontally, vertically, or diagonally
    - Cannot move onto a space occupied by a piece of the same color
    - Can capture any piece of the opposing color by moving onto its space

The king is a player's most important piece. When a player's king is at risk of being captured by the opponent, it is "in check." The player with a king in check must immediately make a move to protect their king, either by:

- Moving the king to a safe position
- Moving a piece between the king and the threatening piece, blocking the attack (note that since knights can move through other pieces, this is not an option when the king is being threatened by a knight)
- Capturing the threatening piece

Neither player may make any move that puts their own king in check (or allows their king to remain in check). If a player in check is unable to make any move to protect their king, that causes a "checkmate" and they lose the game, since their king would be captured on the opponent's next turn.

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 8 lines representing the layout of a chessboard during a game of chess. Pieces are represented with

uppercase or lowercase letters as shown above; uppercase letters represent white pieces, and lowercase letters represent black pieces. Unoccupied spaces are represented by a period. White's position is at the bottom of each board; Black's position is at the top of the board.

```
3
..k.....
.PQP....
........
........
........
........
........
..R....K
........
k.......
........
........
......Q.
.....bnr
......PP
.......K
......k.
........
..rr....
.b......
........
........
........
..RKR...
```

Sample Output

For each test case, if a checkmate exists, your program must print a line containing the word "CHECKMATE" and the winning color. If no checkmate exists (neither player is in check, or the player in check is able to move), print NO CHECKMATE instead.

```
CHECKMATE WHITE
NO CHECKMATE
CHECKMATE BLACK
```