Code Quest Problem Packet

2022 Annual International Contest

Saturday, April 30, 2022

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, .cs .cpp, or .py 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

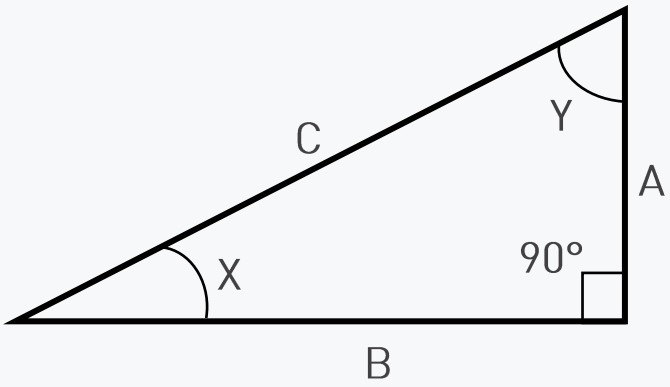
Some problems may require the use of trigonometric functions, which are summarized below. Most programming languages provide built-in functions for $\sin X$, $\cos X$, and $\tan X$; consult your language's documentation for full details. Unless otherwise stated in a problem description, it is *strongly 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 a given value (or values) includes that/those value(s). For example, the range "1 to 3 inclusive" contains the numbers 1, 2, and 3.
- **Exclusive** indicates that the range defined by a given value (or values) does not include that/those values. 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: Null Cipher

Points: 5

Author: Javier Jimenez, Marietta, Georgia, United States

Problem Background

There are two primary methods to obscuring information you wish to keep hidden. Cryptography uses an algorithm or similar process to convert a message to another form, rendering it illegible. Steganography aims to simply hide a message so that a would-be eavesdropper doesn't realize a message actually exists in the first place. Steganography has taken a wide range of forms throughout history: messages written in invisible ink, patterns representing Morse Code knitted into sweaters, and messages shrunk to microscopic size and printed on transparent film have all been used throughout history. One of the simplest forms of steganography is known as the "null cipher."

Problem Description

The null cipher is effective because it appears to be a perfectly harmless message. The secret message - known as a ciphertext - is hidden within another message by adding in a large number of "null" values, either words or letters, which have nothing to do with the original message. Ideally, an eavesdropper would see the message and not realize there was a second message hidden inside, but the intended recipient would know to remove certain words or characters to restore the original message.

Lockheed Martin is working with the National Security Agency to test a slightly different form of null cipher. The NSA intends to embed a message within a string of random characters. Their hope is that an eavesdropper *will* suspect a hidden message, but will assume that the random nature of the message means that it's encrypted using a cipher, and will waste time attempting to break it. In reality, the message will simply be scattered throughout the text string. Any character that is part of the actual message will immediately follow an English vowel; that is, one of the letters a, e, i, o, or u. When those characters occur in the actual message, they will follow a different vowel; the character after them is *not* part of the message. For example, the string below can be read as "hello world":

`fksahnlgueyilfhnalfkjnhdssaokjfhndsfiwaourhnfdjgbalfkjshedfnsf`

Given some sample strings generated by the NSA, design a program that can extract the original messages.

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 lowercase letters.

```
3
fksahnlgueyilfhnalfkjnhdssaokjfhndsfiwaourhnfdjgbalfkjshedfnsf
mkjmnacioudhrieeqwthyiugueresjfgwatfhwghfnhgnffn
elruoqywicwnjksakvfbsgyohuehnghiefhggadfgsfsfs
```

Sample Output

For each test case, your program must print a single line containing the plaintext message extracted from the input string.

```
helloworld
codequest
lockheed
```

Problem 02: Leak Loss

Points: 5

Author: Tai Do, Sunnyvale, California, United States

Problem Background

Your family has an above-ground pool which you'd drained before the winter so it wouldn't freeze over and get damaged. Summer is approaching, and your parents have asked you to use the garden hose to fill it back up again. The pool is filling a lot more slowly than usual, but eventually you get it filled up. When you go to pull the hose back out of the pool, you notice that the drain valve is still open, and spewing water everywhere! You're able to close the valve, but how much water have you just wasted?

Problem Description

You'll need to calculate how much water spilled out of the pool while it was being filled. Since the pool did get filled eventually, you know that the amount of water going into the pool was greater than the amount leaking out of it. As a result, you can use this formula to determine the amount lost:

$$\frac{Volume\ of\ Pool}{(Rate\ of\ Fill - Rate\ of\ Leak)} * Rate\ of\ Leak = Volume\ of\ Waste$$

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 three positive numbers, separated by spaces:

- The volume of the pool, in liters
- The rate at which water was entering the pool, in liters per minute
- The rate at which water was leaving the pool, in liters per minute

```
2
700 10 3
2000 100 20
```

Sample Output

For each test case, your program must print the amount of water that leaked out of the pool while it was being filled, rounded to the nearest liter.

```
300
500
```

Problem 03: The Good Ship Input

Points: 10

Author: Lucas Doty, Orlando, Florida, United States

Problem Background

Enterprise level maintenance applications are prevalent in most industries, and are especially important for all branches of the military. Lockheed Martin provides such applications to military groups around the world. In order to keep aircraft, trucks, ships, and every other sort of vehicle in battle-ready condition, soldiers need to know what parts need to be inspected or maintained at what times. They also need to know when a part might be reaching the end of its operational lifespan so it can be replaced. Keeping track of all of this maintenance work is critical to ensuring missions are successful and accidents are kept to an absolute minimum.

A lot of work goes into building such applications, however. All sorts of files, messaging, and interfaces are being used and sent throughout the layers of logistics application. They can link up to databases to store information and maintain archives. They may need to communicate with other applications to get up-to-date information. Users also need to be able to interface with the application to access all of this data. It's vitally important that users are able to access the information they need, when they need it.

Problem Description

You're working with Lockheed Martin's Rotary and Mission Systems division to build a new maintenance system for the United States Navy. The Navy wants to automate some of the reporting work that needs to be done when a ship is brought into drydock for maintenance. Specifically, they want the ship itself to be able to report what systems are in working order and can be skipped during inspection.

When the ship is brought into drydock, a computer on board the ship will run several automated diagnostic tests on critical systems on board. Once the tests are finished, the computer will transmit a list of those systems that passed their tests to a computer kept in the shipyard. This computer will compare the list of functional systems with a list of systems onboard the ship, to be retrieved from a database. Any systems that appear in the database, but not in the list reported by the ship's computer, need to be reported to maintenance staff for inspection.

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, X, represents the number of systems listed in the shipyard database.
    - The second integer, Y, represents the number of systems reported by the ship's computer. Y will be less than or equal to X.
- X lines containing the names of systems listed in the shipyard database. Names are unique within a test case and may contain letters and spaces.
- Y lines containing the names of systems reported by the ship's computer as being in working order. All names in this section will be duplicates of those stored within the shipyard database.

```
1
5 3
Cannon
Engine
Helm
Deck
Anchor
Engine
Helm
Anchor
```

Sample Output

For each test case, your program must print a list of systems that require inspection, in alphabetical order (case-insensitive), with one system per line.

```
Cannon
Deck
```

Problem 04: Special Treatment

Points: 15

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

Problem Background

Special characters - generally speaking, any character other than a letter, number, or space - can often cause problems in software applications. A string of text with a quote inside of it may be interpreted incorrectly. A backslash in front of a letter could prevent that letter from being read as a letter. In extreme cases, special characters could be used to inject new code into an application, allowing hackers to steal data, or worse! Proper handling of special characters is therefore a major concern for software companies, including Lockheed Martin.

They say computers work on the "garbage in, garbage out" principle... so let's make sure we take out the trash.

Problem Description

Your program will need to read in several lines of text containing a mix of characters. You'll need to remove any character from the string that is neither a letter, number, or space, and print all of the letters, numbers, and spaces that remain.

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 of text to be processed, which may contain any printable ASCII character.

```
3
This “is” amazing!
Can’t you - yes, you - remove all the special characters?
Anything /not/ a letter, numb3r, or _space_ should go.
```

Sample Output

For each test case, your program must print the line of text with all special characters removed.

```
This is amazing
Cant you  yes you  remove all the special characters
Anything not a letter numb3r or space should go
```

Problem 05: Tricky Timecards

Points: 20

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

Problem Background

In any workplace environment, tracking how much time you work is an important factor in making sure you get paid. Large corporations like Lockheed Martin also use timecard information as a basis for estimating the cost of potential contracts; by checking how much time was spent on previous contracts, an educated guess can be made as to how much time will be required for similar work.

Problem Description

Your manager is out on vacation and has asked her assistant to manage some of her responsibilities while she's away. This includes approving the timecards for everyone on your team. Unfortunately, the IT department has just taken the timecard system down for maintenance, and it won't be back online until just before the approval deadline! The good news is the timecard information can be accessed from another system, but it will need to be added together. Your manager's assistant has asked for your help in writing a program that can read this data and summarize it for her, so she can quickly approve everyone's timecard when the system returns.

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 following information, separated by commas:

- A string containing an employee's name, which will contain uppercase and lowercase letters and at least one space. All employee names will be unique.
- Five time values, each in HH:MM format, representing the amount of time worked by that employee for each day of the week (Monday through Friday)

```
3
Peter Gibbons,01:23,04:16,00:59,02:23,00:00
Milton Waddams,08:00,08:00,08:00,08:00,08:00
Bill Lumbergh,08:31,07:59,06:01,08:55,05:30
```

Sample Output

For each test case, your program must print a single line containing the following information:

- The employee's name, as provided in the input
- An equals sign (=)

- The total time the employee worked during the week, in the format “H hours M minutes”, subject to the points below:
    - If the employee worked between 1 (inclusive) and 2 (exclusive) hours, print the word “hour” instead of “hours.”
    - If the number of minutes is equal to 1, print the word “minute” instead of “minutes.”
    - If the number of minutes is equal to 0, omit the number of minutes; instead, simply print the number of hours, without any trailing whitespace.

```
Peter Gibbons=9 hours 1 minute
Milton Waddams=40 hours
Bill Lumbergh=36 hours 56 minutes
```

Problem 06: Find the Missing Sensor

Points: 25

Author: Sowmya Chandrasekaran, Sunnyvale, California, United States

Problem Background

Any time anything is manufactured, it must be tested to ensure that all required parts are in place and in working order. This can include diagnostics of electronic components to ensure that they are able to receive input and provide output correctly. If any errors are detected, they must be fixed before the item is shipped to a customer; it's much easier and less costly to fix problems before an item leaves the manufacturing floor!

Problem Description

A materials engineer working at Lockheed Martin Aeronautics is testing flight sensors being installed on an F-35 fighter jet. The diagnostic test he's running sends a signal to each sensor; the sensors must respond with a number representing their unique ID. Unfortunately, the engineer has noticed a problem; one of the sensors is missing! He's asked your team to write a program to quickly identify the missing sensor.

You'll be given the results of the engineer's diagnostic. This will include a randomly sorted list of the integer ID numbers for each sensor. Each number is unique and will range from 1 to the number of sensors the engineer expected to find (N). One of the numbers between 1 and N will be missing and must be reported to the engineer.

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 single positive integer, **N**, representing the expected number of sensors
- A line containing a list of **N-1** integers, separated by spaces and ranging from 1 to **N** inclusive, representing the ID numbers of working sensors.

```
2
9
5 7 3 2 8 1 4 9
10
7 4 1 10 8 2 9 6 3
```

Sample Output

For each test case, your program must print a single line containing an integer representing the ID number of the missing sensor.

```
6
5
```

Problem 07: Power to Persevere

Points: 25

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

Problem Background

When we look at building an automated machine to travel and explore beyond the current reach of human exploration - whether that be under the sea, into the caldera of a volcano, or somewhere beyond our atmosphere - understanding the limits and capabilities of our device is paramount to the successful execution of the mission. If the machine breaks down in such an extreme environment, you can't simply send a mechanic to repair it. It must be ready to take on any challenge it may face; and its operators need to understand what challenges are beyond its abilities.

NASA is working with Lockheed Martin to design a new lunar rover for future unmanned missions to the Moon. Your team is working on the design of the rover's propulsion systems and is evaluating several possible designs. Before you begin building a prototype, you'd like to evaluate the specifications to see if the designs could possibly work.

Problem Description

The rover's wheels will be powered by a series of servo motors. Each motor is able to spin at a certain speed (measured in revolutions per minute, RPM); each revolution requires a certain amount of power from the rover's batteries. It will take several revolutions of the motor to complete one rotation of the rover's wheels. Your team's task is to analyze data regarding this entire process and determine if the rover's power and propulsion systems are able to get the rover to its destination, and if so, how long it will take to get there.

There are a few formulas which will help you make these determinations:

- Circumference of a circle: $C = \pi d$
  - d = diameter of the circle
- Power (measured in watts): $P = IV$
  - I = Current (measured in amperes)
  - V = Voltage (measured in volts)

For example, let's examine a motor that runs at 12 volts, uses 6 watts of power to complete one revolution, and has a speed of 10 RPM. It takes the motor 6 revolutions to turn a 15 cm-diameter wheel. The rover needs to travel a distance of 5 meters. Each rotation of the wheel allows the rover to move a distance of 47.12 cm; so, over a distance of five meters (500 cm), this requires about 10.61 rotations of the wheel. This in turn requires 63.66 revolutions of the motor, which will take 6.367 minutes. Each revolution of the motor draws 6 watts of power, so the motor will draw a total of 381.96

watts over the course of the mission. Dividing this by the voltage (12 V) shows us that we need a total of 31.83 amps of current to provide that amount of power. Multiplying this by the time of the mission gives us the total energy that must be provided by the power systems, 202.6616 ampere minutes, or 3.3777 ampere hours. As long as the power systems are able to supply this much current in that amount of time, the mission should be a success.

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 seven positive numbers, separated by spaces and representing (in order):

- The diameter of the rover's wheel, in centimeters (cm)
- The number of motor revolutions required to complete a full rotation of the wheel
- The amount of power required to complete one revolution of the motor, in watts
- The speed of the motor, in revolutions per minute
- The available capacity of the power system, in ampere hours
- The voltage requirement of the motor
- The required distance the rover must travel, in meters

```
2
15 6 6 10 4 12 5
12 8 12 10 24 12 8
```

Sample Output

For each test case, your program must print a single line with the following information:

- If the mission is possible with the given parameters:
    - The word "Success"
    - A space
    - The time required for the rover to travel the given distance, in minutes rounded to four decimal places. Include any trailing zeroes.
- If the mission is not possible with the given parameters, the word "Fail"

```
Success 6.3662
Fail
```

Problem 08: DNA Storage

Points: 30

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

Problem Background

The ability to store and retrieve data is a critical part of any computing system. As we develop more and more powerful computer systems, however, this becomes an even greater challenge. In order to keep up with the rate at which our computers are able to process data, we have to develop systems that can store very large amounts of data very quickly. So far, we've made significant progress - the amount of data that can be stored on a single MicroSD card would require entire rooms full of magnetic tapes just half a century ago. However, we have to continue to find more efficient storage methods, and recent studies have found that we may have had the solution inside us all along.

DNA, or deoxyribonucleic acid, is a complex molecule found in the cells of every living thing. DNA contains instructions on how our cells should function; it's effectively life's "programming language." It takes a lot of DNA to describe how a human, plant, or animal should function, and that correlates to a lot of data. For this reason, recent research has tried to investigate ways to use DNA molecules to store and retrieve computing data. Despite the amount of information DNA is able to store, it is still a molecule, and therefore presents an extremely space-efficient means of storing data.

Lockheed Martin is working with a startup biotech company to develop a device that can read (or "sequence") DNA molecules to convert data into binary computer data. Given the structure of a DNA molecule, your device will need to translate and display the information it contains.

Problem Description

DNA consists of two twisted chains of molecules called "nucleotides." When reading information from a DNA molecule, special enzymes are used to unravel a DNA molecule to read the nucleotides contained in a single chain. There are four kinds of nucleotides found in DNA, each typically represented by a single letter: Adenine (A), Thymine (T), Guanine (G), and Cytosine (C). These nucleotides create pairs which join the two chains together to form the complete DNA molecule. A always pairs with T, and G always pairs with C. Since only two pairs are possible, this allows us to create a DNA molecule that represents a binary string, without being concerned about which chain in the molecule gets read.

For this problem, you will be given the nucleotide sequence of one chain of a DNA molecule that contains computing data. You will need to translate the nucleotide bases into binary digits as shown:

Adenine (A) or Thymine (T) = 0

Guanine (G) or Cytosine (C) = 1

Once translated into a binary string, you will need to convert the binary values to ASCII characters. Each character will be represented by seven bits (seven nucleotides), and all characters will be printable. For example, see the DNA sequence below:

G A T C A T A	G C A T C A G	C G A G C T A	G C A C G T A	C G A G C G C
1 0 0 1 0 0 0	1 1 0 0 1 0 1	1 1 0 1 1 0 0	1 1 0 1 1 0 0	1 1 0 1 1 1 1
H	e	l	l	o

A full listing of all US ASCII characters and their binary equivalents is provided on page 4 of this packet.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will consist of a single line of text containing only the characters A, C, G, and/or T. The number of characters in each line will be evenly divisible by seven.

```
2
GATCATAGCATCAGCGAGCTAGCACGTACGAGCGC
CTTATGGCTCATTCTGGATGTACGAATTACCATGTAGCATTATCTAATG
```

Sample Output

For each test case, your program must print the text represented by the binary string contained within the given DNA sequence.

```
Hello
CQ2020!
```

Problem 09: Past Is Prologue

Points: 35

Author: Carlos Sepúlveda, Orlando, Florida, United States

Problem Background

When managing training exercises, organizers of the training event aim for the best outcomes for those participating in the training. Capturing basic details of these training events can often be the deciding factor in whether future training efforts succeed or fail. Knowing basic details like frequency and duration of past events can help maximize scheduling efficiency; knowing the number of attendees of past events can help predict common logistical issues. This is where engineering teams can assist, by developing predictive algorithms that can help organizers make informed decisions with amazing results.

Problem Description

The Orlando Code Quest Junior League Association has been hosting daily events to help teams across Florida prepare for next year's Code Quest and other competitive programming events. They keep detailed records of everyone who attends each event in a database, which stores the following information:

- Data Source ID - A unique alphanumeric identifier for the data record
- Participant - The name of the team or person attending the event
- Session - Either 'Day' or 'Night,' indicating which session of the event was attended
- Event ID - A unique alphanumeric identifier for the event
- Team - Either 'true' or 'false', indicating if the attendee was a team or not

The Association wants to be able to predict how many teams to expect at future events and needs to know what attendance was at each session of past events in order to do so. Your team has been hired to develop a tool that will take in a comma delimited export of their database to collect this information.

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 single positive integer, **N**, representing the number of database records
- **N** lines, each representing a database record, containing the following information separated by commas:
    - The entry's data source ID, a string containing uppercase alphanumeric characters
    - The participant's name, a string containing letters and spaces

    - The session, one of the strings ‘Day’ or ‘Night’
    - The event ID, a string containing uppercase alphanumeric characters
    - A boolean value (‘true’ or ‘false’), indicating if the participant was a team

```
1
4
A1,Code Questers,Day,E1,true
A2,Maintenance,Day,E1,false
A3,LM Peeps,Night,E1,true
A4,Code Questers,Day,E2,true
```

Sample Output

For each test case, your program must print one line for each unique event ID listed in the database records, in alphanumeric order. Each line should contain the following information, separated by commas:

- The event ID, as provided in the input
- The number of teams that attended the Day session of that event
- The number of teams that attended the Night session of that event

```
E1,1,1
E2,1,0
```

Problem 10: Emergency Update

Points: 40

Author: Kelly Reust, Denver, Colorado, United States

Problem Background

Each year, Lockheed Martin employees are encouraged to update their emergency contact information. This information is most often used to warn employees of facility closures due to severe weather or other extreme situations, but can also be used to alert employees to situations they may need to be aware of when travelling, like transportation strikes or notices from local authorities.

It's time for the annual update, and the Human Resources department wants to make sure that they have a record of the changes that were made. Audits of this sort are important to ensure that data is correct; if any data is incorrect, they help investigators track down the source of the errors. As the department's new intern, you've been assigned the task of performing this audit, but you really don't want to have to do it all by hand. Let's write a program to do the work for us!

Problem Description

The HR department has provided you with a list of last year's emergency contact information, and the list of this year's. Most employees will likely leave their information the same, but some may have moved or gotten new phones, and needed to update their records. Additionally, some employees have left or joined the company during the past year, and their records need to be deleted or created, respectively. Your program will need to identify all of these changes and accurately report them.

Each record in the files you have lists the employee's name, address, and phone number. Employee names are unique (for the purposes of this problem). If a name appears in the "old" file but not the "new" file, that shows the employee left Lockheed Martin, and their record should be deleted. Conversely, a name that appears only in the "new" file represents a new employee, whose record should be created. When the same name appears in both files, the address and phone number will need to be compared between the two versions. If either (or both) are different, the record needs to be updated with the new information. If all the information for an employee is the same in both files, no changes were made, and nothing needs to be reported.

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, O and N, representing the number of old and new records, respectively.

- O lines containing last year's emergency contact records. Each line contains the following values, separated by commas:
    - The employee's name, which may contain upper- and lower-case letters and/or spaces.
    - The employee's phone number, which will be a 10-digit positive integer
    - The employee's address, which may contain letters, upper- and lower-case letters, and/or spaces
- N lines containing this year's emergency contact records. Each line contains the following values, separated by commas:
    - The employee's name, which may contain upper- and lower-case letters and/or spaces.
    - The employee's phone number, which will be a 10-digit positive integer
    - The employee's address, which may contain letters, upper- and lower-case letters, and/or spaces

```
1
3 3
John Doe,1234567890,123 Anywhere Street
Jane Doe,9876543210,456 Somewhere Road
Billy Bob Joe,1472583690,789 Nowhere Avenue
Jane Doe,9876543210,456 Somewhere Road
Joe Bob Bill,9638520147,159 Over There Lane
John Doe,1597538462,123 Anywhere Street
```

Sample Output

For each test case, your program must print a list of those employees who had records created, updated, or deleted. One employee should be listed per line, in alphabetical order by name (as presented; first name first). Lines should be formatted as follows:

- If an employee's record was created or deleted, print the employee's name, followed by the phrase " CREATED" or " DELETED", as applicable.
- If an employee's record was updated, print the employee's name, followed by the phrase " UPDATED ", then by "PHONE NUMBER", "ADDRESS", or "BOTH", as applicable.

```
Billy Bob Joe DELETED
Joe Bob Bill CREATED
John Doe UPDATED PHONE NUMBER
```

Problem 11: Counting-out Rhyme

Points: 45

Author: Wojciech Kozioł, Mielec, Poland

Problem Background

Everyone remembers from their childhood some variety of a "counting-out rhyme;" an often-nonsensical rhyme used to select or eliminate one person from a group. In the United States and United Kingdom, these rhymes commonly begin with "eeny, meenie, miney, moe;" a Polish version goes:

Hana, man, mona, mike;
Barcelona, bona, strike;
Hare, ware, frown, vanac;
Harrico, warico, we wo, wac!

As each word is spoken, the speaker points to one person in the group in turn. Upon reaching the end of the rhyme, whomever is being pointed at is eliminated from the group. The rhymer starts the rhyme again, pointing at the next person in line for the first word, and so on until only one person remains.

Problem Description

Everyone knows that the real trick in using these rhymes is knowing who to start with so you don't eliminate yourself. In the Polish rhyme above, there are sixteen separate words. If you're in a group of five people, each person can be given a unique number; naturally, you'll give yourself number 1. If you also start the rhyme with yourself...

1. Hana
2. Man
3. Mona
4. Mike

5. Barcelona
1. Bona
2. Strike
3. Hare

4. Ware
5. Frown
1. Vanac
2. Harrico

3. Warrico
4. We
5. Wo
1. Wac!

Oh no! You're pointing at yourself, and so you're eliminated! That's not what we wanted. But if you start with someone else, they'll be eliminated instead. If you start with #4, they get eliminated in the first round. You'd then start the second round with #5 (the next person in line), eliminating #3; #5 also starts round three, which eliminates #5; then you start the fourth round with yourself, and eliminate #2. You've won!

Given the number of people in your group and the number of words in your counting rhyme, write a program that can determine who you should start counting with in order to guarantee that you are the last person standing.

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 positive integers separated by spaces:

- N, the number of people in the group. Each person is identified by a unique number between 1 and N inclusive; you are #1.
- K, the number of words in your counting rhyme

```
2
5 16
6 8
```

Sample Output

For each test case, your program must print the number of the person with whom you should start counting in the first round to guarantee that you are not eliminated in any round.

```
4
5
```

Problem 12: Morse Code

Points: 50

Author: Richard Green, Whiteley, Hampshire, United Kingdom

Problem Background

Morse Code, named for its creator Samuel Morse, is a widely recognized method of encoding text through a series of dots and dashes. This was originally created in the 1800's for use with the telegraph, a device that used electrical pulses to transmit messages, but has also been used with sound, light, and radio signals as well.

When using a telegraph, the operator would push a button to send an electrical signal. The signal would remain active as long as the button was held; releasing the button would end the signal. When transmitting a message, the operator would represent a dot by holding down the button for one unit of time, and a dash by holding down for three units of time. Spaces between dots and dashes within a character would be represented by releasing the button for one unit of time; spaces between characters would be represented by a three-unit delay, and between words a seven-unit delay.

Problem Description

The original Morse Code was designed for efficiency; the amount of time required to transmit each English letter was inversely proportional to the frequency with which it was used. For example, the most common letter in the English language, E, was represented by a single dot. Conversely, Q, one of the least common letters, required two dashes, a dot, and another dash.

With modern computers, efficiency is less of a concern; you and your friends are more concerned about eavesdroppers. You've agreed to create a custom version of Morse Code to use when sending messages back and forth. All you need to do is write a program that can read in the Morse alphabet you'll be using, then encrypt and decrypt messages using that code.

Messages in Morse Code will be represented in text as a series of periods (for dots), dashes, and spaces. Each dot and dash within a letter should be separated by a single space. Letters within a word should be separated by three spaces. Words should be separated by seven spaces.

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:

- 26 lines representing the Morse Code alphabet to use for this test case, listed in alphabetical order. Each of these lines will begin with an uppercase letter, followed by a space, followed by a

combination of periods (.), dashes (-), and spaces, representing the Morse Code representation for that letter.
- A line containing a message written in uppercase letters and spaces that must be encoded using the provided Morse Code.
- A line containing a message written in the provided Morse Code that must be decoded.

```
1
A . -
B - . . .
C - . - .
D - . .
E .
F . . - .
G - - .
H . . . .
I . .
J . - - -
K - . -
L . - . .
M - -
N - .
O - - -
P . - - .
Q - - . -
R . - .
S . . .
T -
U . . -
V . . . -
W . - -
X - . . -
Y - . - -
Z - - . .
HELLO WORLD
. .       . - . .   - - -   . . . -   .       - . - .   - - -   - . .   .       -
- . -   . . -   .   . . .   -
```

(Note that the encrypted message above represents only one line of text; it is too long to fit on this page. See the provided sample input file for a more accurate representation.)

Sample Output

For each test case, your program must print:

- A line containing the now-encrypted Morse Code message
- A line containing the now-decrypted English message

```
. . . .   .   . - . .   . - . .   - - -       . - -   - - -   . - .   . - . .   -
. .
I LOVE CODE QUEST
```

(Note that the encrypted message above represents only one line of text; it is too long to fit on this page. See the provided sample output file for a more accurate representation.)

Problem 13: What to Do?

Points: 55

Author: Gary Hoffmann, Denver, Colorado, United States

Problem Background

Computers have a limited number of resources available; they only have so much memory, disk space, and computing power available. The computer's operating system is responsible for managing competing demands for these resources, and often have to make decisions about what software gets priority use of those resources.

You're working with Lockheed Martin Space to develop a satellite for the National Oceanic and Atmospheric Administration (NOAA) to conduct various experiments high above Earth's atmosphere. Your team has been asked to design a priority scheduler that will determine what the satellite should be doing at any point in time.

Problem Description

The satellite's computer breaks down time into computing cycles, starting with cycle 1. Each task will take a certain number of cycles to complete, and once a task is completed, it should not be executed again. Tasks have a priority associated with them; at the start of each computing cycle, the scheduling algorithm should instruct the satellite to work on the highest priority task (higher priorities have larger values) from amongst those it has not yet completed, possibly causing it to switch between tasks. In the event two tasks have the same priority, run the one that appears first in the list of tasks. Tasks also have a time constraint, represented by a cycle number. A task may not be worked unless the current cycle number is greater than or equal to the task's time constraint.

Your scheduling algorithm will need to provide a report of what the satellite will be doing during each cycle for a given number of cycles. If the satellite is not able to work any incomplete tasks due to time constraints, the satellite should perform the "Wait" task for one cycle, and evaluate the situation again at the start of the next cycle. It may not be possible to complete all tasks in the time allotted.

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 comma, representing:
  - C, the number of cycles the scheduling algorithm should schedule
  - T, the number of tasks the satellite should be instructed to perform
- T lines, providing information about the tasks to be completed by the satellite. Each line will contain the following information, separated by commas:

- A string providing a unique description of the task, which may contain letters and spaces
- A positive integer representing a priority level for the task
- A positive integer representing the task's time constraint, the earliest cycle number at which the task can be started
- A positive integer representing the number of cycles this task will take to complete

```
1
10,7
Execute Experiment A,11,4,2
Reorient to Target A,4,1,2
Capture Imagery,5,1,3
Station Keeping,10,1,2
Transmit Results,20,8,1
Reorient to Target B,12,4,1
Execute Experiment B,2,2,3
```

Sample Output

For each test case, your program must print C lines containing information about the satellite's tasking during each computing cycle. Each line should contain the cycle number (starting with 1), a comma, and the description of the task to execute during that cycle. If there is no task to execute, print "Wait" in place of the task description.

```
1,Station Keeping
2,Station Keeping
3,Capture Imagery
4,Reorient to Target B
5,Execute Experiment A
6,Execute Experiment A
7,Capture Imagery
8,Transmit Results
9,Capture Imagery
10,Reorient to Target A
```

Problem 14: You Can Depend on Me

Points: 60

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

Problem Background

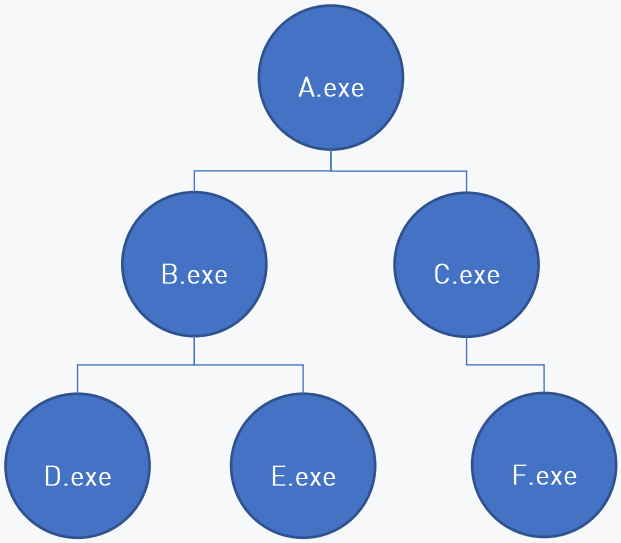
DevSecOps - an abbreviation of development, security, and operations - focuses on making everyone accountable for the security of an application, from the start of its development to its release and beyond. As a DevSecOps engineer at Lockheed Martin, you're responsible for maintaining the software that your team relies on to develop your application. This infrastructure includes tools that constantly evaluate the application for bugs, vulnerabilities, and other issues that should be resolved before delivering an application to the customer.

Unfortunately, the programs that make up this infrastructure are tightly coupled to each other, and when one process crashes, all the processes supporting that process must also be restarted. Also unfortunately, this happens on a somewhat regular basis as your coworkers continue to develop the application and add new features. Your task is to implement a program that monitors the running processes for a failure, then once one occurs, restart that program and all of its dependencies.

Problem Description

Your development infrastructure consists of a number of related processes. When a program becomes unresponsive and needs to be rebooted, all of its dependent processes need to be restarted first. If those processes have dependencies of their own, they need to be restarted as well, and so on.

For example, in the diagram below, a total of five programs are supporting the system. B.exe depends on both D.exe and E.exe to function; if B.exe crashes, D.exe and E.exe will also need to be restarted, before B.exe itself is restarted.



If A.exe were to stop responding, the entire infrastructure would need to be restarted, since all other programs are below it in the dependency tree. Since D, E, and F are the lowest level dependencies, they will again need to be restarted first; B and C can then be restarted before finally restarting A. When multiple programs exist at the same dependency level (as with D, E, and F), programs should be restarted in alphabetical order by name.

Your program will need to read in a list of program dependencies and a list of failure events that must be

handled. For each event, it will need to report the programs that must be restarted, and in what order to restart them. Programs may have more than one dependency, as shown above, and more than one program may depend upon another. There will be no circular dependencies; that is, no program will ever depend upon itself, no matter how indirectly.

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:

- A line containing two positive integers, separated by a space, representing:
  - D, the number of dependencies
  - E, the number of failure events
- D lines containing program dependency information. Each line consists of two strings, separated by a space. The program named by the first string depends on the program named by the second string. All program names will contain lowercase letters and will end with the '.exe' extension.
- E lines containing failure event information. Each line will contain the name of a program, previously listed in the dependency list, that has failed and must be restarted.

```
1
5 2
a.exe b.exe
a.exe c.exe
b.exe d.exe
b.exe e.exe
c.exe f.exe
b.exe
a.exe
```

Sample Output

For each test case, your program must print the commands to be executed for each failure event, one per line, in the order in which they are to be executed. First, restart commands consist of the phrase "restart", followed by a space and the name of the program to restart. These should be followed by a single 'exit' command to complete the recovery process.

```
restart d.exe
restart e.exe
restart b.exe
exit
restart d.exe
restart e.exe
restart f.exe
restart b.exe
restart c.exe
restart a.exe
exit
```

Problem 15: Codebreaker Returned

Points: 65

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

Problem Background

In 2021, we presented a problem in which the National Security Agency had asked Lockheed Martin to help develop a frequency analysis database for cracking substitution ciphers. The Agency was so impressed with your work, they've asked you to continue your efforts!

As a reminder, substitution ciphers encrypt messages by replacing letters in the original message with different letters, in an effort to hide the content of the message. A simple substitution cipher might replace each letter with the next one in the alphabet; for example, "cat" would become "dbu." These are generally regarded as weak ciphers, however, because in each language, certain letters appear more commonly than others. Vowels are particularly common because of how important they are to the language. These common letters serve as guideposts to anyone trying to break such a cipher; if you know the original message was in English, you can safely assume that the most common letter is likely to be 'E'. This method of breaking ciphers is called "frequency analysis."

More advanced ciphers take this into account and go to greater lengths to obscure the text; however, more advanced frequency analysis can circumvent this. Rather than looking at individual letters, looking at pairs or triplets of letters (known as digraphs and trigraphs) can provide even greater insight and help with the identification of individual letters.

Problem Description

As mentioned above, the National Security Agency has asked Lockheed Martin to expand the frequency analysis database created in 2021 to include an analysis of digraphs and trigraphs. A digraph is a pair of letters that appear next to each other in the same word; a trigraph is a group of three letters in the same word. As with letters, certain digraphs and trigraphs appear much more commonly in certain languages, and certain combinations never appear at all. For example, you'll never see the letters 'qxz' appear next to each other in any English word, but the trigraph 'the' is the most common - largely because 'the' is the most common word in the English language.

When performing your analysis, keep in mind that digraphs and trigraphs do not cross word boundaries, and all punctuation and numbers should be removed prior to starting your analysis. For example, the phrase 'code quest' includes the digraphs 'co', 'od', 'de', 'qu', 'ue', 'es', and 'st'. It does not contain the digraph 'eq'; even though those letters appear next to each other, they are separated by a space. The same applies to trigraphs; that phrase includes 'ode' and 'que', but not 'deq' or 'equ'.

As with the 2021 problem, your team will be analyzing a large amount of text in order to identify the relative frequencies of the digraphs and trigraphs you find within that text. The relative frequency of a term can be calculated using this formula:

$$Relative\ frequency = \left(\frac{Occurrences\ of\ term}{Total\ number\ of\ terms\ of\ the\ same\ type}\right) \times 100\%$$

As noted, keep the count of all digraphs and the count of all trigraphs separate when performing this calculation.

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 the results of your analysis, first printing a single line for each digraph identified in the text, then printing a single line for each identified trigraph. Each list should be presented in alphabetical order. Each line should contain the following information:

- The digraph or trigraph, in uppercase
- A colon (:)
- A space
- The relative frequency of the given term within the text provided in the test case, relative to other terms of the same type, rounded to three decimal places and including any trailing zeroes.
- A percent sign (%)

The sample output is too large to print in full here; instead, a small selection of lines from the sample output is presented in order to demonstrate the format. For the full sample output, please download the file from the contest website.

```
AB: 0.840%
AG: 0.840%
```

[...additional lines for AI through WN...]

```
YR: 0.840%
ZY: 0.840%
ABO: 1.124%
AGE: 1.124%
```

[...additional lines for AIN through VAN...]

```
VER: 2.247%
YRE: 1.124%
```

Problem 16: Synaptic Server

Points: 70

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

Problem Background

Lockheed Martin has been contracted by the United States Army to build a new self-driving troop transport. Your team has been assigned to build a street sign identification system for the transport. Your system is required to take a low-resolution photograph of a street sign and output a string indicating the type of sign in the picture; one of:

- STOP_SIGN
- YIELD
- LANE_ENDS
- SPEED_LIMIT
- CROSSWALK

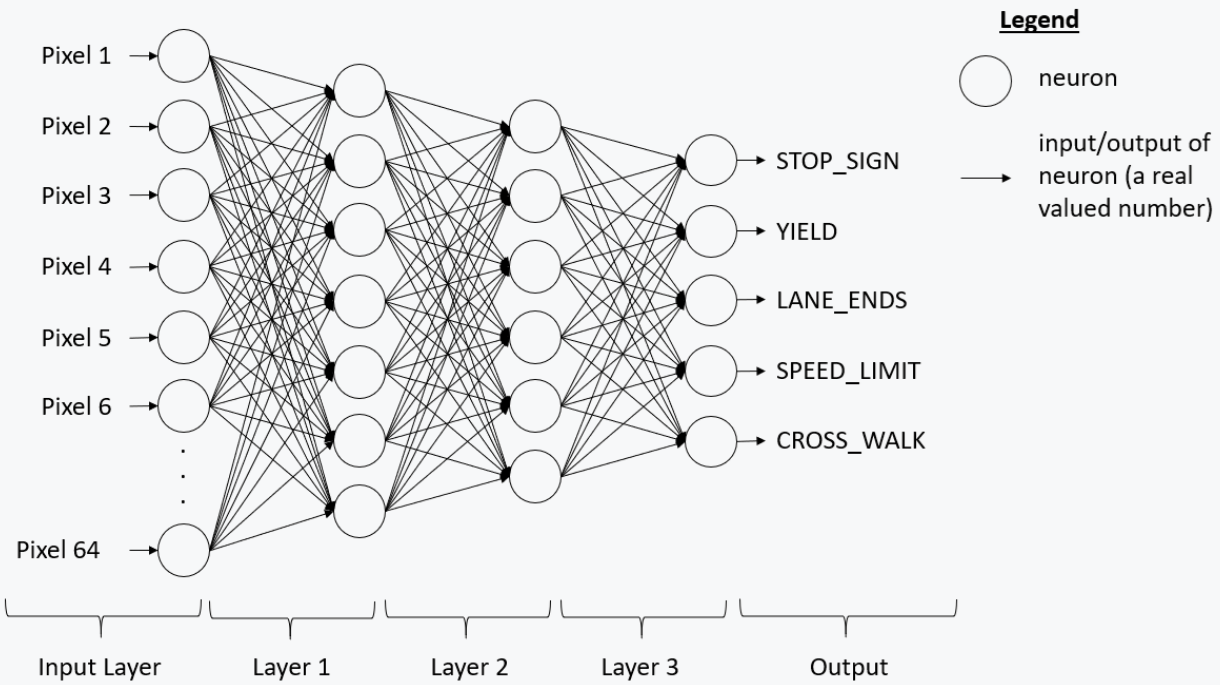
This is harder than it sounds, because no two pictures will be the same. The position of the transport, the time of day, poor weather, or even graffiti on the sign can prevent an accurate identification. However, your algorithm must be able to overcome these problems, because a wrong answer could be disastrous!

Writing a traditional function to solve this problem would be nearly impossible, but you know a team of humans could identify signs very easily. This creates an ideal setup for a class of algorithms called supervised machine learning, where a computer "learns" an unknown function using a large amount of known sample data.

Problem Description

Your lead engineer suggests using a neural network to solve this problem. A neural network is a machine learning algorithm loosely based on the structure of the brain. There are two main components to a neural network: neurons and layers. Neurons are the fundamental computing unit within a neural network, and are grouped into a series of many layers.

Each neuron in a neural network contains a list of weights ($\vec{w}$) and a single bias value (b). These numbers make neurons configurable, allowing the computer or engineer to adjust those values as the computer learns from existing data. Each neuron is responsible for producing a single "real valued number" (any positive, negative, or zero number); the outputs from all neurons in a layer are then collected into a single list and sent as the inputs to each neuron in the next layer. The image on the next page illustrates the layout of a neural network and how these inputs and outputs are passed between layers.



Each neuron uses the following formula to calculate its output value.

$$y = \max\left(0, b + \sum_{i=0}^{N} x_i w_i\right)$$

In this formula:

- y is the neuron's output value
- $\max(\quad)$ is a function that returns the larger of the two numbers given to it
- b is the neuron's bias value (different for every neuron)
- Σ is a summation operator; it indicates that the portion of the equation after it must be repeated for each value of i from 0 through N inclusive, and that all of those values must then be added together to form a single number.
- N is the number of inputs for the neuron; this is equal to the number of neurons in the previous layer
- $\vec{x}$ is a list (containing $x_0, x_1, \ldots, x_N$) with the input values for the neuron (the outputs from all neurons in the previous layer, in order)
- $\vec{w}$ is a list (containing $w_0, w_1, \ldots, w_N$) with the neuron's weights for each input (different for every neuron)

This formula could also be written as:

$$y = \max(0, b + x_0 w_0 + x_1 w_1 + \cdots + x_N w_N)$$

All neurons in the same layer share the same set of inputs; however, their unique weight and bias values ensure that every neuron produces a different output.

For the troop transport's neural network, each neuron in the first layer will receive as input a list of 64 numbers between 0.0 and 1.0 inclusive; each number in the list represents the grayscale value of a pixel in an 8-by-8 pixel image of a street sign. These neurons will produce outputs to be passed to the next layer, which will produce outputs to be passed to the next layer, and so on. The final layer will contain only five neurons, representing the five types of street sign to be identified, as follows:

1. STOP_SIGN
2. YIELD
3. LANE_ENDS
4. SPEED_LIMIT
5. CROSSWALK

The outputs produced by each of these five neurons represent how confident the network is that the corresponding sign is the correct answer. Your system needs to determine which output value is the highest and return that string as the final answer.

Fortunately, your lead engineer was able to locate data from several pre-trained neural networks; your team has to test all of these networks to see how well they perform.

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 greater than or equal to 3, L, representing the number of layers in the neural network.
- A line containing L positive integers, separated by spaces, representing the number of neurons in each layer of the network. The last integer in this list, representing the output layer, will always be 5. Other integers may range from 6 to 128, inclusive.
- L lines containing the decimal bias and weight values of each neuron in the layer. Values will be separated by spaces. For each neuron in the layer, one weight value will be provided for each input, followed by the neuron's bias value. For example, a layer with five neurons that receives six inputs would list values as follows:

  $$w_{0,0}\ w_{0,1}\ w_{0,2}\ w_{0,3}\ w_{0,4}\ w_{0,5}\ b_0\ w_{1,0}\ w_{1,1} w_{1,2}\ w_{1,3}\ w_{1,4}\ w_{1,5}\ b_1 \ldots\ w_{4,4}\ w_{4,5}\ b_4$$

- A line containing 64 decimal numbers between 0.0 and 1.0 inclusive and separated by spaces, representing the pixel values to be given as the inputs to all neurons in the first layer.

The sample input for this problem is too large to publish in the Problem Packet. Please download the sample input from the contest website.

Sample Output

For each test case, your program must print a line containing the name of the street sign with the highest confidence level calculated by the neural network. Street signs should be printed exactly as shown in the description above.

```
SPEED_LIMIT
YIELD
```

Problem 17: Get Out the Vote

Points: 75

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

Problem Background

Voting is a critically important part of a functioning democracy. By voting in an election, you're giving your input to how you want to be governed. Even if your chosen candidates don't win the election, you're still sending a message to politicians of every party about the principles you support. Anyone with the right to vote should be sure to exercise that right at every opportunity; this helps to ensure a better future for everyone.

Unfortunately, democracies and elections aren't perfect. It often turns out that even if many candidates are running in an election, only two or three of them stand any real chance of winning. If a voter doesn't fully support any of these candidates, they may feel forced to choose between the "lesser of two evils," or that it's not actually worth voting at all. In order to combat these feelings, some jurisdictions are switching to a new system of running elections, called Ranked Choice Voting. Under this system, a voter who votes for a losing candidate still gets a say in which of the other candidates gets elected.

Problem Description

Lockheed Martin is working with a state government to implement a new electronic voting system that utilizes Ranked Choice Voting. Your team has been asked to work on the algorithm that will tally the final votes. While several different versions of Ranked Choice Voting exist, the one the state is using works as follows.

Voters list candidates in order of preference; the candidate they support most is listed first, followed by their second choice, and so on. Once all the ballots have been cast, the results are tallied using the first choice listed on each ballot. If any candidate has a clear majority of votes - half of the total number of ballots, plus one - they win the election. Otherwise, the candidate that garnered the least number of votes is disqualified, and a second tally begins; in the event of a tie for last place, all candidates with the lowest vote count are disqualified together. Any ballot listing the disqualified candidate as the first choice now contributes its second choice to the tally. Once again, if any candidate has a clear majority, they win; otherwise, one or more candidates are disqualified, and a third tally is performed, using second- or third-choices as needed. This continues until a clear winner emerges.

For example, consider an election in which four candidates are running. After the first tally, the results are as shown below:

Candidate	First-Choice Votes	Percentage
Candidate A	10	40%
Candidate B	7	28%
Candidate C	5	20%
Candidate D	3	12%

None of the candidates reached more than 50% of the vote, so nobody wins... yet. Candidate D got the fewest votes, and so is disqualified. Of the 3 people that voted for Candidate D, 2 marked Candidate B as their second choice, and 1 marked Candidate C. After the second tally, the result is:

Candidate	1st-Choice	2nd-Choice	Total	Percentage
Candidate A	10	0	10	40%
Candidate B	7	2	9	36%
Candidate C	5	1	6	24%
~~Candidate D~~	~~3~~	~~0~~	~~0~~	~~Disqualified~~

There's still no winner here, so we do one more elimination now, removing Candidate C from the running. Anyone who voted for Candidates C or D as their first (or second) choice now must resort to their second (or third) choice. In this scenario:

- The person who listed Candidate D as their first choice and Candidate C as their second choice chose Candidate A as their third choice, which is now counted.
- 2 people who voted for Candidate C as their first choice chose Candidate D as their second choice. Since Candidate D is disqualified, they resort to their third choice: Candidate A.
- The other 3 people who voted for Candidate C first chose Candidate B as their second choice.

Candidate	1st-Choice	2nd-Choice	3rd-Choice	Total	Percentage
Candidate A	10	0	3	13	52%
Candidate B	7	5	0	12	48%
~~Candidate C~~	~~5~~	~~1~~	~~0~~	~~0~~	~~Disqualified~~
~~Candidate D~~	~~3~~	~~2~~	~~0~~	~~0~~	~~Disqualified~~

After this count, Candidate A has a clear majority now, with 52% of the vote. They are declared the winner.

Your algorithm must read in a number of ballots cast for an election and determine the winner.

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:
  - B, the number of ballots cast in the election
  - C, the number of candidates running in the election
- B lines representing each of the ballots cast. Each line will contain C characters, each representing a candidate, listed in order of preference.

```
1
25 4
ABCD
ABDC
ACBD
ACDB
ADBC
ADCB
ABCD
ADCB
ACDB
ABDC
BACD
BADC
BCAD
BCDA
BDAC
BDCA
BACD
CDAB
CDAB
CBAD
CBDA
CBAD
DBAC
DBCA
DCAB
```

Sample Output

For each test case, your program must announce the winner of the election, by printing the sentence "Candidate X won with P% of the vote after T tallies", where:

- X is the uppercase letter used to identify the candidate in the input.
- P% is the percentage of the final vote, rounded to one decimal place and including any trailing zeroes.

- T is the number of tallies that had to be conducted to reveal a winner.

```
Candidate A won with 52.0% of the vote after 3 tallies
```

Problem 18: This Is Rocket Science

Points: 75

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

Problem Background

Lockheed Martin and the UK Space Agency are working together to develop a new system for deploying CubeSats - small satellites often used for scientific research. The Small Launch Orbital Manoeuvering Vehicle - or SL-OMV - can hold up to 6 CubeSats at a time and launch them at the optimal times and positions for their respective missions. This makes launching CubeSats considerably easier, particularly since CubeSats are usually too small to have any sort of propulsion systems of their own. That said, launching the satellites is still not an easy task.

Problem Description

Your team is working with the UK Space Agency to create a program to determine how much fuel must be loaded onto the SL-OMV for each mission. The navigational systems onboard the vehicle allow it to move as though on a two-dimensional grid. On each mission, the vehicle will have up to six CubeSats which must each be launched at a specific location on that grid while the vehicle is at a complete stop relative to the grid. This ensures that each CubeSat is placed into the appropriate orbit for its mission. However, the vehicle must also launch each CubeSat at a specific time to ensure that the CubeSat's orbit won't interfere with that of another satellite. Collisions in space are a very serious concern, and can cause catastrophic damage not just to the colliding objects, but also to other satellites in nearby orbits.

The SL-OMV has four thrusters, each allowing it to accelerate along either the X or Y axis. Each thruster provides a constant acceleration of 1 m/s^2 while it is being fired. Keep in mind that the vehicle is operating in space - with no air resistance or other friction to slow it down, the vehicle will continue to move at a constant rate of speed once its thrusters are shut off. This means you must fire the opposite thruster(s) to slow the vehicle down or cause it to reverse direction.

For example, let's say that the SL-OMV must launch a CubeSat at coordinates (3,3) within 5 seconds. It starts at coordinates (0,0) moving at 0 m/s (note that all speeds shown here are relative to the grid; in reality, orbital speeds are measured in *kilometers* per second). Getting to the launch coordinates is easy; a short burst on both the positive X and positive Y thrusters will cause the vehicle to start moving in the right direction; once it approaches the launch coordinates, bursts on the negative thrusters of equal duration will cause it to stop on target. However, a short 0.1 second burst on each thruster wouldn't move the vehicle very quickly; there's no way it would arrive in time to be able to launch the CubeSat without interfering with other satellites.

We'll need to get the vehicle moving a bit faster in order to launch in time. The CubeSat must be launched after 5 seconds. Once we get the launch vehicle up to speed, we'll have to expend the same amount of time and fuel to slow it back down again. This means we can focus simply on the first half of the trip; the speeding up. Once we determine how much fuel is needed to reach halfway, we can simply double our calculations to get the total amount of fuel required to speed up *and* slow down.

Let's begin our calculations. If we're only concerned about getting halfway, we need to be at coordinates (1.5,1.5) after 2.5 seconds. Part of this time (t_1 seconds) will be spent firing our thrusters; once we get to the necessary speed, we can stop firing, and simply coast for the remaining time (t_2 seconds). Since our total journey will take 2.5 seconds, $t_2 = 2.5 - t_1$.

The distance travelled by a moving object can be calculated using this formula:

$$x = \frac{1}{2}at^2 + ut + x_0$$

Here, x represents distance, a represents acceleration, t represents time, and u represents the object's initial velocity (before it started accelerating). x_0 represents the starting position of the object. While we're accelerating, a is equal to 1 m/s^2, and u is 0. Once we stop accelerating, a becomes 0, and v becomes however fast we're moving (note that we've switched to v, as now it's our *final* velocity). This means we'll need to use two equations to accurately represent the distance we've travelled:

$$x_1 = \frac{1}{2}a{t_1}^2$$

$$x_2 = vt_2$$

$$x = x_1 + x_2$$

$$t = t_1 + t_2$$

How fast will we be moving once we shut off the thrusters, however? As it turns out, we don't really need to know; velocity can be calculated by multiplying your acceleration by the amount of time you spend accelerating. In other words, we can replace v with at_1 and not have to worry about its actual value.

With all of this knowledge, we can finally combine all of our knowledge into a single equation that we can solve to determine how long the thrusters need to be fired.

$$x = x_1 + x_2$$

$$x = \frac{1}{2}a{t_1}^2 + vt_2$$

$$x = \frac{1}{2}a{t_1}^2 + at_1(2.5 - t_1)$$

$$x = \frac{1}{2}a{t_1}^2 + 2.5at_1 - a{t_1}^2$$

$$a = 1; x = 1.5$$

$$1.5 = 2.5t_1 - 0.5{t_1}^2$$

$$0 = -0.5t^2 + 2.5t - 1.5$$

We've managed to remove all of the other variables, but this equation isn't something we can solve on its own. By rearranging it into the format shown above, the equation is now something called a quadratic equation; an equation of the form $0 = ax^2 + bx + c$. The equation below can be used to solve this type of equation:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$t = \frac{-2.5 \pm \sqrt{2.5^2 - (4 * -0.5 * -1.5)}}{2 * -0.5}$$

$$t = \frac{-2.5 \pm \sqrt{6.25 - 3}}{-1}$$

$$t = 2.5 \pm \sqrt{3.25}$$

$$t = 0.6972 \; or \; 4.3027$$

This equation always gives two answers, but the interesting thing about it is you can always pick whichever answer makes sense for your situation. In this case, 4.3 seconds would be far too much time to fire the thrusters; we'd overshoot the launch coordinates and ruin the mission. On the other hand, firing the thrusters for 0.6972 seconds fits very nicely into our timeframe, and should allow us to reach the launch coordinates on time.

Keep in mind, though, this only covers half the trip; we also have to fire the opposite thrusters for 0.6972 seconds in order to slow down again. We've also only been focused on movement along the X axis during these calculations; the calculations would need to be repeated for movement along the Y axis. As it turns out, we're moving the same distance along both axes, so the numbers end up being the same. This results in a final thruster burn time of 2.7888 seconds (0.6972 seconds in each direction) to go from our starting position of (0,0) to our launch coordinates of (3,3) in exactly five seconds.

Again, your team will need to calculate how much fuel (measured in seconds of burn time) will be required in order to bring the SL-OMV launch vehicle to each of the provided launch coordinates in the specified amount of time, coming to a complete stop each time.

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 information about the initial state of the launch vehicle. This information includes the following values, separated by spaces:
    - A positive integer less than or equal to 6, S, representing the number of CubeSats on board
    - An integer representing the starting X coordinate (in meters)
    - An integer representing the starting Y coordinate (in meters)
- S lines containing information about the launch parameters for each CubeSat on board the launch vehicle, listed in the required launch order. This information includes the following values, separated by spaces:
    - An integer representing the launch X coordinate (in meters)
    - An integer representing the launch Y coordinate (in meters)
    - A non-negative decimal value representing the number of seconds the vehicle must take to travel from its previous location to these coordinates

```
2
2 0 0
3 3 5.0
5 5 3.0
2 0 0
1 7 10.0
5 3 6.0
```

Sample Output

For each test case, your program must print the minimum amount of fuel required to launch the CubeSats in the order listed with the given launch parameters. Fuel amounts should be measured in seconds of burn time, rounded to two decimal places, and include any trailing zeroes.

```
6.79
4.77
```

Problem 19: Trust, But Verify

Points: 80

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

Problem Background

A key issue in software design and development is the concept of data validation. How do you know if the input data you are receiving has been tampered with? One of the primary mechanisms to accomplish this is by using a hash algorithm.

A hash algorithm is a way of condensing any amount of data into a smaller byte array of a specific size. These are one-way functions, so the original data cannot be reconstructed from the hash alone; additionally, they are deterministic, meaning any given block of data will always produce the same hash. While hashes aren't quite unique - there's an infinite number of ways to put together data, but a non-infinite number of hash values - it's nearly impossible to intentionally create a set of data that results in the same hash as another. These properties make these hash functions ideal for data validation.

Once a hash has been calculated for any particular set of data, that hash can be published as a "checksum." Anyone can re-run the same hash algorithm on the source data, and if the result doesn't match the provided checksum, that proves the data has been tampered with.

Problem Description

Your team is working with Lockheed Martin's Corporate Information Security office (LMCIS) to implement a new automated security process for verifying documents. Your program will receive a string of text representing a document being downloaded from Lockheed Martin's servers. It must calculate a hash of that data, then compare that hash against the one provided by a different server. If the hash matches, everything is fine; if not, you'll need to flag the error so Lockheed Martin employees know not to open that document - it's been tampered with!

LMCIS has developed a new hash algorithm for this purpose. (Note that this is actually a bad idea in reality; you should always use previously published and cryptographically secure hash algorithms whenever one is needed to ensure the best security.) The hash algorithm works using the following process:

1. Convert the given text string to binary by replacing each character in the string with the binary equivalent shown in the ASCII table in the reference information, retaining any leading zeroes. For example, 'A' would be replaced with '1000001', and '.' would be replaced with '0101110'.
2. Add additional '1' bits to the end of the binary string until the length is evenly divisible by 128.
3. Split the resulting binary string into "chunks" of 32 bits each.

4. Initialize variables, called A, B, C, and D, to the 32-bit binary equivalents of the following numbers:
    a. A = 792,250,721
    b. B = 683,117,105
    c. C = 1,215,387,974
    d. D = 1,767,829,900
5. For each chunk (from step 3), calculate new values for A, B, C, and D as follows. M represents the current chunk; N represents the index of that chunk (first chunk has index 0, second has index 1, etc.).
    a. S = (B XOR D) AND (C OR (NOT B))
    b. S = A + S
    c. S = M + S
    d. Left rotate S by (N modulo 32) bits
    e. Set A = D, D = C, C = B, and B = S
6. Concatenate the final values of A, B, C, and D into a single binary string (in that order)
7. Convert the result into a 32-character hexadecimal string by replacing every four bits with the corresponding hex character (0000 = 0, 1111 = F, etc.)

In case you're not familiar with the various operations described in step 5:

- A OR B compares each bit in A and B in turn. If at least one bit in A or B is '1', the corresponding bit in the result is also '1'. (e.g. 1100 OR 1010 = 1110)
- A AND B compares each bit in A and B in turn. If both bits in A and B are '1', the corresponding bit in the result is also '1'. (e.g. 1100 AND 1010 = 1000)
- A XOR B compares each bit in A and B in turn. If exactly one bit in A or B is '1', the corresponding bit in the result is also '1'. (e.g. 1100 XOR 1010 = 0110)
- NOT A inverts the value of each bit in A. (e.g. NOT 10 = 01)
- A + B adds the values of A and B together, then truncates the result to match the lengths of A and B. (e.g. 1111 + 1010 = 1001)
- A left rotation of X bits moves the X most significant (leftmost) bits to the end of the number. (e.g. a left rotation of 1 bit for 101010 results in 010101; 2 bits results in 101010.)

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 string of text to be hashed, up to 2000 characters long. This line may contain any character listed in the US ASCII table in the reference information.
- A line containing 32 uppercase hexadecimal characters representing the checksum hash to compare against.

```
2
This is a sample string for which you will need to calculate a hash.
45F1D2C216714CA65BE85211A364B2DB
Once calculated, compare the hash to the given hash and see if they match.
C2935B67EE5204A671D231E94676C101
```

Sample Output

For each test case, your program must print a single line with the word "TRUE" if the hash you calculate matches the provided checksum value, or "FALSE" otherwise.

```
TRUE
FALSE
```

To assist with your debugging efforts, the text for the second test case should produce the hash value shown below. Please note that this is not part of the expected output:

```
43A442826ED0CCB0E89B4160E76087B3
```

Problem 20: Plink

Points: 85

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

Problem Background

Plink is a common carnival game in which a player drops a ball or disc into a slot at the top of a board filled with pegs arranged in offset rows. The ball rolls down the board, bouncing off the pegs as it drops, causing it to change course as it falls. Eventually the ball will fall into one of several buckets positioned at the bottom of the board. Carnival workers assign each bucket a point value, and the goal of the game is to try to get the ball to fall into the bucket worth the most points.

While targeting a specific bucket is the goal of the game, actually targeting a specific bucket is very difficult. The bouncing of the ball causes it to move almost randomly; each time the ball hits a peg, it has a nearly-equal chance to move to either the right or left side of the peg. As a result, the buckets in the middle of the board tend to catch the ball more often, because there are more paths leading to them, and thus a higher chance that the ball will land there. Buckets near the edges have fewer paths, and so catch the ball less frequently. As a result, the buckets near the edges tend to have a higher value than those in the middle.

You're working with a gaming company to test a new form of the game of Plink, a version that requires more skill and eliminates some of the randomness. There are three key differences to this version of the game:

- The ball can only be started in the middle of the board, rather than anywhere across the top.
- The round pegs have been replaced with flippers that can be tilted left or right by the player, allowing them to direct the course of the ball.
- Each peg has a point value; as the ball falls, the player adds the value of each peg the ball hits to their score. This is then added to the value of the bucket in which the ball landed to determine the player's final score.

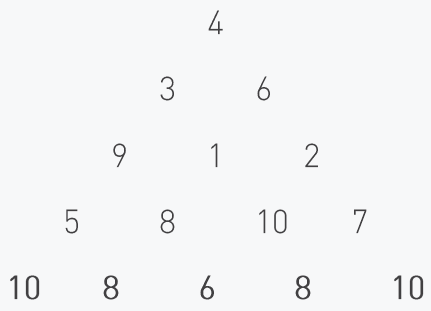
As a result of the changes in format, the player can manipulate the path of the ball to maximize their score. Aiming for a high-value bucket on the edge of the board may not earn the best score, if the only path leading to it contains only low-value pegs. In order to get a high score, a player must find the optimal path for the ball as it falls through the board.

Problem Description

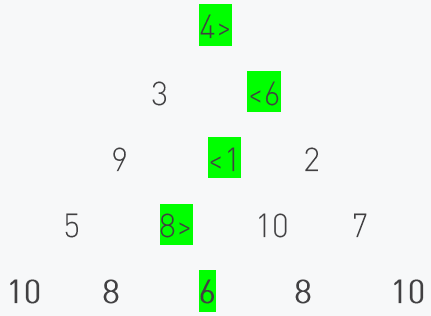
You must design a program that can find the optimal path for a given Plink board; that is, the path that is worth the greatest number of points given the specific arrangement of point values and bucket values. Your path will be expressed as a series of letters - L or R - indicating if the ball should move to

the left or right, respectively, of each flipper it hits as it falls. The ball will only hit one flipper on each level of the board, so the number of directions given will be equal to the number of levels in the board.

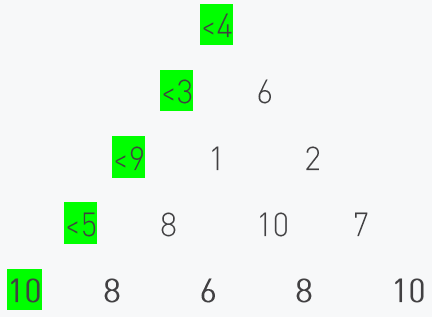
For example, consider the board below. Each number in the triangle represents the point value of the flipper in that position. The bold numbers along the bottom represent the values of the buckets at the bottom of the board.

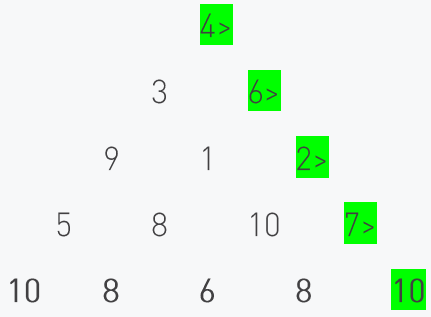


Despite the small size of this board, there are 16 possible paths through it. The path "RLLR," for example, results in a score of 25:

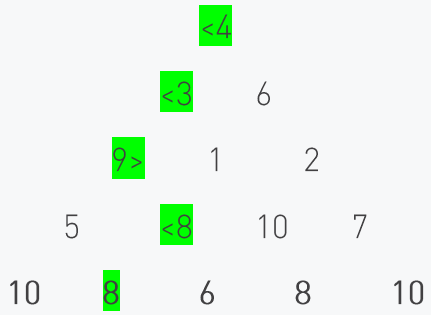


This is not the optimal path for this board. It lands in a low-value bucket in the middle of the board, and also hits a flipper worth only one point. In traditional Plink, it's best to aim for the edges of the board; here, that would be achieved with the paths "LLLL" and "RRRR."





The all-left path earns a score of 31 points, and the all-right path earns a score of 29 points. These are both better than the path we looked at earlier. However, neither is the optimal path. There is one path that earns 32 points; LLRL:



Even though we lose two points by landing in a different bucket, we gain three points by aiming for the 8-point paddle in the middle, rather than staying on the edge.

Your task is to write a program that can calculate the path that earns the maximum possible score for a given Plink board. Be warned, however; these examples covered a very small board consisting of only four rows of paddles. Your team is hoping to design much larger boards, in order to increase the game's difficulty. Each additional row of paddles increases the number of possible solutions by another power of two. Your program must be certain it's found the single best path out of all possibilities.

As a reminder, your program must complete execution within two minutes, or it will be marked as incorrect (even if it would have eventually generated the correct response).

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 rows of paddles in the Plink board.
- X lines, containing the integer point values of the paddles in the corresponding row of the table in order from left to right. Values are separated by spaces, and lines will contain a number of values equal to the row number of the board (the first row is row 1, and will contain 1 value).
- A line containing X+1 integers separated by spaces representing the point values of the buckets at the bottom of the board in order from left to right.

```
1
4
4
3 6
9 1 2
5 8 10 7
10 8 6 8 10
```

Sample Output

For each test case, your program must print a line containing:

- A string with X uppercase letters L or R, representing the optimal path through the given Plink board, as described above.
- A space.
- An equals sign.
- A space.
- An integer representing the number of points obtained from that path.

```
LLRL = 32
```

Problem 21: Shopping Spree

Points: 90

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

Problem Background

Online shopping has become a massive industry over the past two decades. From major corporations like Amazon to small businesses working out of their owner's basements, there are now very few things you can't order online and expect to have delivered to your doorstep a few days later. The digital "shopping cart" concept has even expanded to other applications, and now is a common, intuitive way to view and organize anything you've selected on a website.

Most shopping cart interfaces share two common features: users may only view or modify the contents of their own cart, and items in a user's cart are not purchased until they finish "checking out." This differs from a physical shopping cart like you may see in a grocery store; there, a shopper physically removes an item from the shelf, making it unavailable to other customers. Online, simply having an item in your cart usually doesn't place any sort of reservation on those items. Another person could place the same item(s) in their cart and check out before you have a chance to do so, potentially leaving those items out of stock.

Problem Description

The Lockheed Martin site you're working at is adding a new automated convenience store to allow employees to buy snacks and drinks throughout the day. Employees can select what they want from a website, and equipment in the store will drop those items into a locker for pick up. Your team has been assigned to build the website. As with any website, the site you're designing will need to process each user's commands as they arrive. You need to write a program to handle that processing. However, since multiple users can be connected at one time, the commands from one user may be mixed in with those of others.

This is a new concept, and so the store's inventory isn't very large, and it's possible the store may sell out of items rather frequently. Your program will also be provided with the store's inventory at the start of each day, and should manage that inventory as orders come in. Customers should not be allowed to add any items to their cart that are already out of stock.

Once a customer checks out, the program should total up the cost of all items they have in their cart and remove those items from the inventory. If any items went out of stock between being added to a user's cart and that user's checkout, the user should be notified that those items are not available, and should not be charged for them. The user will be able to return to the site at a later time and attempt to order the items again once they are restocked.

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:
  - I, the number of types of items in the inventory
  - C, the number of commands received to the website from customers
- I lines listing items held in the inventory. Each line will contain three values, separated by spaces:
  - A positive integer indicating the number of items in inventory.
  - A text string containing the name of the item. This may contain uppercase letters, lowercase letters, and underscores (_)
  - A positive decimal value indicating the cost of a single item of this type.
- C lines listing commands received from customers. Each command will begin with a positive integer representing the customer's ID; all commands received from a particular customer will begin with the same ID value. The remainder of the command will include several values, separated by spaces. Possible commands include:
  - ADD <ITEM> <QTY> - Used to add items to the cart. <ITEM> will be the name of the item as it appears in the inventory report, and <QTY> will be replaced with a positive integer representing the number of items to add. If the requested number plus the number of those items already in the customer's cart is equal to or less than the number currently in the inventory, add that number of items to the customer's cart, but do not remove them from the inventory at this time. Otherwise, the command should be rejected.
  - REMOVE <ITEM> <QTY> - Used to remove items from the cart. <ITEM> will be the name of the item as it appears in the inventory report, and <QTY> will be replaced with a positive integer representing the number of items to remove. If the requested number and type of items exists in the customer's cart, those items should be removed from the cart. Otherwise, the command should be rejected.
  - CHECKOUT - Used to finalize the customer's order. The contents of the user's cart should be checked against the current inventory. For any items where the customer cannot receive the full requested amount, give the customer none of those items; instead, remove that item from the cart and report the shortage. All remaining items should then be removed from the inventory, and the total cost of those items reported to the user.

```
1
3 8
2 Candy_Bar 1.50
3 Soda_Bottle 1.60
2 Cheese_Pack 2.50
1 ADD Candy_Bar 1
2 ADD Candy_Bar 2
1 ADD Soda_Bottle 3
1 REMOVE Soda_Bottle 1
2 CHECKOUT
1 CHECKOUT
3 ADD Candy_Bar 1
3 ADD Cheese_Pack 2
```

Sample Output

For each test case, your program must print the results of each command received in the order they are received, as follows:

- ADD commands:
    - If the command is successful, print a single line of the format "Added <QTY> <ITEM> to customer <ID>'s cart"
    - If the command is rejected, print a single line of the format "Not enough <ITEM> for customer <ID>"
    - In both cases, replace <QTY> with the number of items added, <ITEM> with the name of the item, and <ID> with the customer's ID number.
- REMOVE commands:
    - If the command is successful, print a single line of the format "Removed <QTY> <ITEM> from customer <ID>'s cart"
    - If the command is rejected, print a single line of the format "Customer <ID> does not have that many <ITEM>s"
    - In both cases, replace <QTY> with the number of items added, <ITEM> with the name of the item, and <ID> with the customer's ID number.
- CHECKOUT commands:
    - For each item where there is insufficient stock to meet the customer's request, print a line of the format "Out of stock of <ITEM>". If multiple items have insufficient stock, print them in alphabetical order.
    - Finally, print a line of the format "Customer <ID>'s total: $<COST>", replacing <ID> with the customer's ID number, and <COST> with the total cost of all items remaining in the customer's cart, rounded to two decimal places and including any trailing zeroes.

Once all commands are processed, your program must also print one line for each item remaining in inventory, in alphabetical order, in the format "<ITEM> - <QTY>", replacing <ITEM> with the name of the item and <QTY> with the quantity remaining in inventory. Omit items that are out of stock.

```
Added 1 Candy_Bar to customer 1’s cart
Added 2 Candy_Bar to customer 2’s cart
Added 3 Soda_Bottle to customer 1’s cart
Removed 1 Soda_Bottle from customer 1’s cart
Customer 2’s total: $3.00
Out of stock of Candy_Bar
Customer 1’s total: $3.20
Not enough Candy_Bar for customer 3
Added 2 Cheese_Pack to customer 3’s cart
Cheese_Pack - 2
Soda_Bottle - 1
```

Problem 22: Calling All Firefighters

Points: 100

Author: Dr. Leon Clark, Melbourne, Victoria, Australia

Problem Background

In a disaster situation, maintaining open lines of communication with everyone involved - both first responders and regular civilians caught in the disaster - is important to limit casualties and the amount of damage. Unfortunately, geographic features like mountains can make communication difficult. Many modern means of communication rely on radio signals, which can be blocked by large mountains or other features that block a direct line of sight between a signal tower and a communication device.

Problem Description

Lockheed Martin is working with the Australian Defence Force to set up a means of quickly establishing new communication relays in the event of another round of disastrous wildfires. The ADF plans to provide Lockheed Martin with a cross-section of a topographical map showing the layout of terrain in the impacted area. This map will include numbers showing the proposed locations of a communications tower (marked with a number 0) and several command posts (marked with numbers 1 through 9). Your team has been asked to design an algorithm which can determine which of the command posts are viable, based on the location of the communications tower.

The map is extremely low resolution, and so each cell within the map represents a one square kilometer area. Each area is considered to be completely filled either with solid ground (#) or open air (a space or number). In order for a command post to be in a viable location, firefighters must be able to raise a communications antenna in the center of that location; this antenna must have a direct, unobstructed line of sight to the tower. The transmitters for both antenna and tower will be located in the exact center of their respective areas (half a kilometer from the edges of the cell drawn on the map). If a straight line drawn from those two points comes into contact with the ground at any point, the signal from the tower will be at least partially blocked, which will interfere with communications and make the command post's location not viable.

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, as follows:
  - H, representing the height of the map in rows (minimum 4)
  - W, representing the width of the map in characters (minimum 30)

    - C, representing the number of command posts that will be displayed on the map (minimum 2)
- H lines, each containing up to W characters, representing a cross section of a topographical map. Each line may contain any of the following characters:
    - #, representing solid ground, which blocks radio signals.
    - A space, representing open air. Note that input lines will not contain trailing whitespace; if a line contains fewer than W characters, you may assume that any missing characters after the last visible character are intended to be spaces.
    - A number 0 (zero), representing the proposed location of a communication tower. There will be exactly one 0 in each test case.
    - Any number from 1 (one) to C inclusive, representing the proposed location of a command post. There will be exactly one of each relevant number in each test case.

```
2
6 45 4
                                4
            0                  ####
           ####               #######
    #############   2       ############
    3 ################   ################  1
#############################################
5 45 4
            1                  4####
           ####               #######
    #############   2       ############
    3 ################   ################  0
#############################################
```

Sample Output

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

- If at least one command post is in a viable location, print the list of the numbers representing those locations, in increasing order, separated by spaces.
- If no command posts are in viable locations, print the phrase “No viable locations”.

```
2 4
No viable locations
```

Problem 23: aMAZEing

Points: 100

Author: Anonymous Volunteer, Orlando, Florida, United States

Problem Background

Mazes have been of interest to the human race for years. The English word "maze" comes from an old English word meaning delirium or delusion. While mazes and labyrinths are not really the same, they are frequently used synonymously. A labyrinth is usually many winding, curved passages, while mazes are more like a puzzle and have many dead ends. We have ancient Greek mythology with heroes escaping from monsters in a labyrinth. The English built hedge mazes in the gardens of many of their castles. Today, many farmers will build mazes in corn fields after the harvest.

Solving a maze can be fun and that's exactly what this challenge involves. Can you find the shortest path through a simple rectangular maze?

Problem Description

You will be provided with some rectangular mazes and will have to write a program to read the maze diagram, then determine the shortest path through the maze, and report how many "cells" were traversed to move from entrance to exit. Moves are only made horizontally and vertically, and of course may not be made through walls.

Each "cell" within the maze is represented as a 3-by-4 horizontal rectangle of characters, bounded on each corner by a plus sign (+). Each cell shares its edges with its neighbors; that is, the right edge of one cell is also the left edge of the next cell. Dashes (-) and pipes (|) are used between the plus signs to indicate if a cell's edge contains an impassible wall; otherwise, these characters will be spaces, indicating an open space you can pass through. The center of each cell will always contain two spaces. See the example cells below:

<pre>+--+--+
| | |
+--+--+</pre>
	<pre>+--+
	   |
	+ +</pre>

Two fully closed cells, with walls on all sides	A cell with the left and bottom edges open

Each maze will have one entrance and one exit, which will both appear along an outer edge of the maze (and may be located on the same side). They will be represented by replacing a cell's edge with one of the following sets of characters:

- Two lowercase letter v's will replace dashes to represent an entrance along the maze's top edge, and/or an exit along the maze's bottom edge.

- Two carets (^^) will replace dashes to represent an entrance along the maze's bottom edge, and/or an exit along the maze's top edge.
- A right angle bracket (>) will replace a pipe to represent an entrance along the maze's left edge, and/or an exit along the maze's right edge.
- A left angle bracket (<) will replace a pipe to represent an entrance along the maze's right edge, and/or an exit along the maze's left edge.

Each maze may have more than one viable path from the entrance to the exit; again, your goal is to find the shortest path, and report how many cells you had to step through along the way.

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, representing the height of the maze diagram in lines, H, and its width in printed characters, W, respectively.
- H lines, each W characters long, representing the maze diagram. These lines may contain dashes (-), plus signs (+), pipes (|), spaces, carets (^), lowercase letter v's, and/or angle brackets (> or <), as described above.

To avoid breaking across multiple pages, the sample input is provided on the next page.

```
2
11 16
+vv+--+--+--+--+
|        |     |
+--+  +--+  +--+
|  |     |     |
+  +--+  +--+  +
|              |
+  +--+--+  +--+
|     |        |
+--+  +  +--+  +
|           |  >
+--+--+--+--+--+
11 31
+--+--+--+--+--+--+--+--+--+--+
<        |     |     |     |  |
+--+  +--+  +--+  +  +  +  +  +
|  |     |        |     |     |
+  +--+  +  +--+--+--+--+--+  +
|  |        |     |           |
+  +--+--+--+  +  +  +--+  +--+
|     |        |     |        |
+--+  +  +--+--+  +--+  +--+  +
>           |     |     |     |
+--+--+--+--+--+--+--+--+--+--+
```

Sample Output

For each test case, your program must print the number of cells traversed when moving from entrance to exit.

```
9
31
```

Problem 24: The Daily Grind

Points: 110

Author: John Worley, Fort Worth, Texas, United States

Problem Background

The United States and many other countries make use of a 40-hour work week in professional settings. However long an employee works each day, their schedule rounds out to about 40 hours over each seven-day period. Exactly how that 40 hours is reached, however, tends to vary, and the differences can drive the poor accountants in the payroll office absolutely bonkers.

Lockheed Martin has recently switched to a 4/10 work schedule - employees are asked to work only four days a week (Monday through Thursday), but work ten hours each day. However, this schedule is flexible, and employees can work with their managers to find a schedule that works best for them. Human Resources needs help sorting through the different schedule requests to determine how much work will actually get done each month.

Problem Description

A variety of different work schedules have cropped up over the last few decades, and four of these have seen fairly common use at Lockheed Martin recently:

- A **40-hour** schedule is a traditional work week; employees work 8 hours each day, 5 days a week (Monday through Friday)
- A **4/10** schedule condenses the work week; employees work 10 hours each day, 4 days a week (Monday through Thursday)
- A **9/80** schedule works a bit differently; employees work 9 hours each day, 4 days a week (Monday through Thursday), plus 8 hours every other Friday. This works out to a total of 80 hours over a two-week period, averaging to 40 hours per week. There are two versions of this schedule, 9/80A and 9/80B, which determine which Fridays are taken off. For the purposes of this problem:
    - A **9/80A** schedule will work the first Friday of each calendar year (without consideration of other years)
    - A **9/80B** schedule will take off the first Friday of each calendar year (without consideration of other years).

Lockheed Martin's accounting system tracks the number of working days by period, rather than calendar months. Each period is centered around a calendar month, but may not start or end on the same dates. Each period ends on the last Friday of the calendar month; the next Monday, then, is the start of the next period. For example, the period for April 2022 runs from Monday, March 28th to Friday, April 29th; the period for May 2022 runs from Monday, May 2nd to Friday, May 27th.

The Human Resources department has asked your team to create a calculator that determines the correct number of working days for each schedule in the period containing any given date. When a given date falls on a weekend (Saturday or Sunday), use the previous Friday to determine the correct period. Don't worry about any holidays, vacations, or other potential time off; part of the reason HR wants this is to determine when that time off should be scheduled!

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 date in DD/MM/YYYY format.

```
3
14/03/2022
24/04/2022
29/05/2022
```

Sample Output

For each test case, your program must print a single line containing the number of working days for each work schedule within the period containing the given date. Print each schedule's working day count as an integer, separated by spaces, in the following order:

- 40-hour
- 4/10
- 9/80A
- 9/80B

```
20 16 18 18
25 20 23 22
20 16 18 18
```

Problem 25: Take Me Out to the State Machine

Points: 130

Author: Louis Ronat, Denver, Colorado, United States

Problem Background

Baseball is a very popular sport in the United States and several other countries. A long-standing practice amongst baseball enthusiasts is to complete a scorecard as the game progresses. This scorecard records most details of the game so that it can be completely reconstructed afterwards. This is possible because a game of baseball proceeds as a series of states, with a pause in between each action taken by the offensive team. This contrasts with a sport like basketball or soccer where play will progress for minutes without a break, making a full reconstruction of events very difficult.

This play-and-pause behavior makes a baseball game function as a state machine; a construct in which a set of actions result in one of several distinct "states." If these actions are recorded and repeated by another identical state machine, it will end up in the same state as the original machine. Here, the scorecard serves as the record of actions taken within the game.

Problem Description

You will need to read and parse a baseball scorecard in order to reconstruct the events that took place during that game. As part of this process, you will need to determine the final score of the game, as well as show how the field looked at specific points during the game.

First, a quick review of the game of baseball; if you're already familiar with the game, you can skip this paragraph, as it won't provide any information relevant to the problem that you don't already know. Baseball is played by two teams on a field known as a diamond, due to the arrangement of four "bases" around the field. Over the course of nine "innings", teams take turns on offense (or "at bat") and defense (or "in the field"). The away team is always at bat first, in the "top" of the inning; the home team is at bat second, in the "bottom" of the inning. Players on the team currently at bat take turns standing at home base attempting to hit a ball thrown by the pitcher of the team in the field. If the batter misses or doesn't attempt to hit the ball, they are awarded a "strike" or "ball" based on several factors. If the batter hits the ball, they become a "runner" and attempt to run counter-clockwise around the field, to first, second, and third base, then back to home base. Each runner earns one point for his team when he crosses home base without being tagged out by the team in the field. The teams switch roles once three players on the team at bat are out.

(Continue reading here if you skipped the paragraph above.) Baseball scorecards show the events that occurred while a team is at bat. They are presented as a table, with each row representing a different inning and each column showing the actions taken by each player during the inning. These actions are represented by acronyms for the various plays that can be made during a game. For this

problem, we will use a simplified set of acronyms that don't cover all possible plays or all possible outcomes from those plays. These include:

Offensive plays:

- BB - Walk / Base on Balls. The batter has earned four balls and is able to walk to first base unopposed. Other runners already on the field do not advance, except as needed to allow the batter to move to first base. (e.g., a runner on first base would walk to second base, but a runner on second base would not move unless someone was also on first base.)
- S - Single. The batter hits the ball and is able to advance to first base. Other runners will advance one base unless tagged out.
- D - Double. The batter hits the ball and is able to advance to second base. Other runners will advance two bases unless tagged out.
- T - Triple. The batter hits the ball and is able to advance to third base. Any other runners will advance to home base and score unless tagged out.
- HR - Home Run. The batter hits the ball out of the field and is able to cross home plate and score along with any other runners.

Outs:

- B - Bunt. The batter hits the ball a short distance and is tagged out before he can reach first base. Other runners already on base will advance one base.
- FC - Fielder's Choice. A fielder tags out the batter or one of the runners. Other runners (including the batter) will advance one or more bases.
- DP - Double Play. The team in the field tags out two of the runners (one of whom may be the batter). Other runners (including the batter) will advance one or more bases if they are not also tagged out.
- K - Strike Out. The batter has earned three strikes and is out without reaching a base. Any runners already on base do not advance.
- PF - Pop Fly. The batter hits the ball up high, but it is caught before touching the ground, resulting in an out. Any runners already on base do not advance.

Each cell within the scorecard's table indicates the position of that player on the field after each play during the inning; first base is shown at the right of the cell, second base at the top, third base on the left, and home base at the bottom. If the player made it to a base, their initial position will be marked with the acronym for the offensive play they made. As the plays made by teammates allow the player to advance around the bases, a number will be displayed at each of the player's positions representing the number of their teammate that allowed the advance. For example, see the cell below showing Player 1's actions:

```
|   2   |
|     S|
|   4   |
```

In this case, Player 1 hit a single and was able to advance to first base (indicated by the "S" on the right side). Player 2 later came to bat and allowed Player 1 to advance to second base (indicated by the "2" at the top). Player 3 didn't allow Player 1 to advance, and so his number doesn't appear; however, Player 4 allowed Player 1 to advance to home base and score (indicated by the "4" at the bottom). Notice that we don't know what Players 2-4 did on their turns at bat just from this cell; this is documented in their own sections of the scorecard.

When a batter or runner is out, the acronym representing the cause of their out is shown in the center of the diamond. For example:

```
|       |
|  DPBB|
|       |
```

This player was able to reach first base with a walk; however, before he was able to each any other bases, he was tagged out in a double play on a teammate's turn at bat.

Each cell in the scorecard represents multiple states during the course of the game. The second example shows two states; one in which the batter was able to walk to first base, then another when he was tagged out and removed from the field. This doesn't preclude the possibility of other states in between those states; another player may have struck out between the walk and double play, without affecting the position of the runners. As a result, you have to look at the diagrams for other players to get a full understanding of what happens during each inning.

Your program must read in a scorecard for each team in order to reconstruct the events that took place during the game. You will then be asked to show the state of the field at specific points during the game, as well as the final score of the game.

Sample Input

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

- A line containing two positive integers separated by spaces; respectively:
    - X, the number of innings played during the game. This number will be between 1 and 9 inclusive.
    - Y, the number of points in the game for which you must show the state of the field.
- Two scorecards laid out over multiple lines. The first scorecard represents the away team and the top of each inning; the second scorecard represents the home team and the bottom of each inning. The format of each scorecard is as follows:
    - A line containing a header for the scorecard, formatted as follows:

        - A space followed by a pipe (|)
        - Nine cells containing information about each player, formatted as follows:
            - A space
            - An integer, 1 through 9 inclusive, representing the player's batting number. Players will be listed in their batting order, from 1 to 9.
            - A space
            - Two uppercase letters representing the player's initials
            - A space followed by a pipe (|)
    - A row divider consisting of 65 dashes (-).
    - X table rows, each consisting of four lines of text and containing ten cells. The first cell in each row contains spaces except for the second line, which will contain a positive integer between 1 and X inclusive. The other cells will contain spaces, numbers, and uppercase letter acronyms, arranged as described in the problem description above. Cells are divided by a column of pipes (|), and every row ends with a divider line consisting of 65 dashes (-).
- Y lines listing the points during the game for which you must display the state of the field. These lines contain the following information, separated by spaces:
    - The phrase "TOP OF" or "BOTTOM OF" if the point during the game is during the away team's at bat or the home team's at bat, respectively.
    - A positive integer from 1 to X inclusive, representing the number of the inning
    - The word "PLAYER"
    - A positive integer from 1 to 9 inclusive, representing the number of the player about to bat. The point in time to represent is just as the indicated player is stepping up to bat, before they have taken the action shown in the scorecard.

To avoid breaking across pages, the sample input begins on the next page.

```
1
3 5
  | 1 CF | 2 FB | 3 LF | 4 SS | 5 CA | 6 TB | 7 RF | 8 SB | 9 PI |
-----------------------------------------------------------------
  |      |   D  |      |      |      |      |      |      |      |
1| 2  BB| 4    |   K  |  DP S|  DP  |      |      |      |      |
  |   4  |      |      |      |      |      |      |      |      |
-----------------------------------------------------------------
  |      |      |      |      |      |      |      |      |      |
2|   K  |      |      |      |      |  PF  |    BB|      |  PF  |
  |      |      |      |      |      |      |   8  |  HR  |      |
-----------------------------------------------------------------
  |      |      |      |      |   6  |      |      |      |      |
3|      |   K  |  PF  | T    |    BB|     S|  PF  |      |      |
  |      |      |      |   6  |      |      |      |      |      |
-----------------------------------------------------------------
  | 1 FB | 2 RF | 3 CA | 4 SB | 5 TB | 6 LF | 7 SS | 8 CF | 9 PI |
-----------------------------------------------------------------
  |      |   D  |      |      |      |      |      |      |      |
1|   K  |      |    BB|      |  PF  |   K  |      |      |      |
  |      |   4  |   4  |  HR  |      |      |      |      |      |
-----------------------------------------------------------------
  |      |      |      |      |      |      |   8  |   9  |      |
2|   B  |     S|   K  |      |      |      | 9   S| 1   S|   B  |
  |      |      |      |      |      |      |   1  |   2  |      |
-----------------------------------------------------------------
  |      |      |      |      |      |      |      |      |      |
3|      |      |      |      |      |      |      |      |      |
  |      |      |      |      |      |      |      |      |      |
-----------------------------------------------------------------
TOP OF 1 PLAYER 4
BOTTOM OF 1 PLAYER 2
TOP OF 2 PLAYER 6
BOTTOM OF 2 PLAYER 3
TOP OF 3 PLAYER 7
```

Sample Output

For each test case, your program must print the following information:

- Y diagrams of the baseball field, representing the state of the game as the indicated player comes up to bat during the indicated inning. Each diagram must be formatted as follows:
    - A line containing two spaces followed by the uppercase initials of the player on second base. If second base is unoccupied, print two underscores in place of the initials (__).
    - A line containing a space, a forward slash (/), two spaces, and a back slash (\)

    - A line containing the uppercase initials of the player on third base, two spaces, and the uppercase initials of the player on first base. If either base is unoccupied, print two underscores in place of the respective initials (__)
    - A line containing a space, a back slash (\), two spaces, and a forward slash (/)
    - A line containing two spaces followed by the uppercase initials of the player currently at bat.
    - A line showing the score at that point in the game, including:
        - A number showing the home team's score
        - A dash (-)
        - A number showing the away team's score
    - A line showing the number of outs at that point during the inning, formatted as an integer followed by a space and the word "OUT"
    - Remember that no lines should contain any trailing spaces.
- A line showing the game's final score, formatted as described below. The score after all provided innings is the final score, even if the number of innings played is less than 9.
    - If one team had a higher score than the other:
        - The word "FINAL:" followed by a space
        - The word "HOME" or "AWAY", depending on which team had a higher score
        - A space followed by the phrase "TEAM WINS" and another space
        - The home team's score, printed as an integer
        - A dash (-)
        - The away team's score, printed as an integer
    - If both teams had the same score:
        - The phrase "FINAL: TIE GAME" followed by a space
        - The home team's score, printed as an integer
        - A dash (-)
        - The away team's score, printed as an integer

```
  FB
 /  \
CF  __
 \  /
  SS
0-0
1 OUT

  __
 /  \
__  __
 \  /
  RF
0-1
1 OUT

  __
 /  \
__  __
 \  /
  TB
3-1
0 OUT

  __
 /  \
__   RF
 \  /
  CA
5-3
2 OUT
  CA
 /  \
__   TB
 \  /
  RF
5-4
2 OUT
FINAL: HOME TEAM WINS 5-4
```