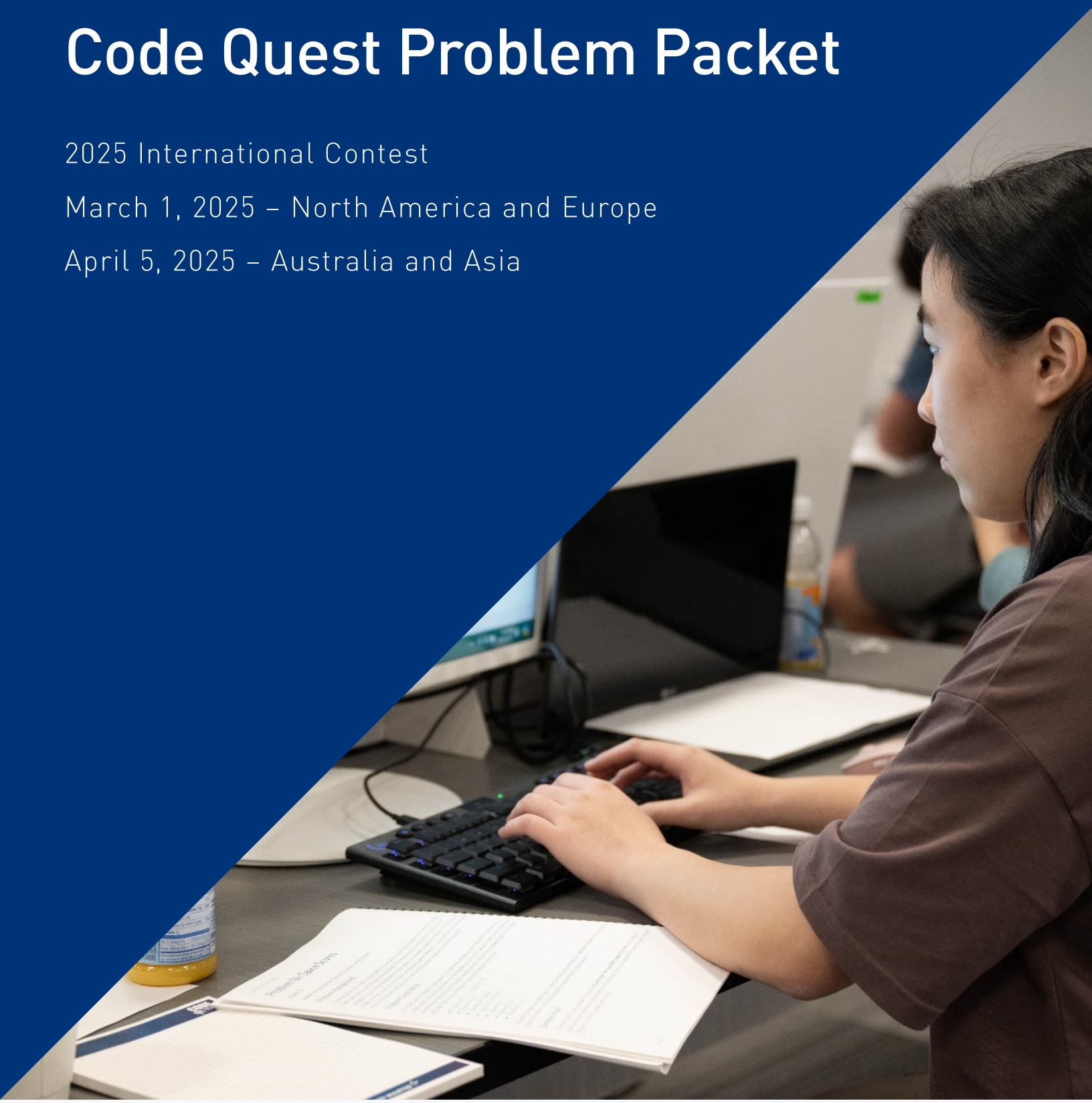
Code Quest Problem Packet

2025 International Contest

March 1, 2025 – North America and Europe

April 5, 2025 – Australia and Asia



CODE QUEST®

LOCKHEED MARTIN

Table of Contents

In Memoriam

Code Quest was created in 2012, and since then we've been proud to support thousands of students around the world achieve their goals and begin successful careers in software engineering. This program would not be possible without the support of hundreds of Lockheed Martin volunteers who offer their time to test problems, handle registrations, set up event spaces, and much more. This past year, we lost two of our most dedicated volunteers, Taylor Little and Gary Hoffmann. Their contributions helped shape Code Quest into what it is today, and we are deeply grateful for their years of service. They will be dearly missed, but their legacy will live on.

Taylor Little played a pivotal role in organizing the first Code Quest event in 2012 in Fort Worth, Texas. She continued to lead the program until her passing, serving as the site lead for Fort Worth, and later as the product owner for Code Quest. Her bold spirit and unwavering dedication not only drove the program forward but also helped expand it into the thriving initiative it is today. Her passion inspired countless others to carry on Code Quest's torch, ensuring her legacy lives on in every future competition.

Gary Hoffmann was a valued member of the problem development team. He contributed many hours helping to test problems and suggested many problems to use in our contests, including two in this year's Problem Packet. Based in Denver, he was a supportive and helpful judge in every year that Code Quest was held there.

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.

Can we access the internet during the competition?

Please see the section regarding Internet Access later in this packet. Certain websites should not be accessed and may be blocked if at an on-site location.

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 contact us at code-quest.gr-eo@lmco.com, and we'll be happy to send them on.

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 away from zero" method of rounding unless explicitly stated otherwise 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 away from zero" rounding, numbers are rounded to the nearest integer. For example:

- 1.49 rounds down to 1
- 1.51 rounds up to 2

The "half away from zero" 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

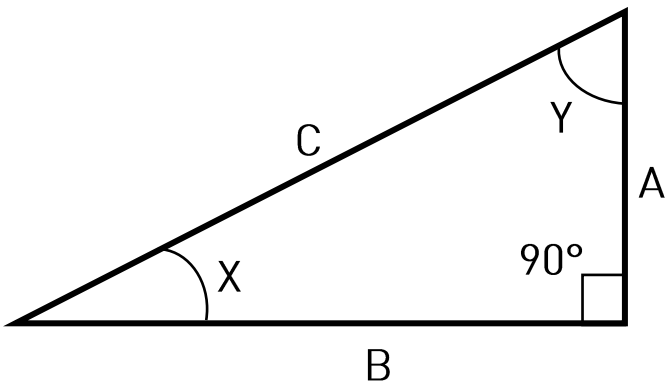
Some problems may require the use of trigonometric functions, which are summarized below. Most programming languages provide built-in functions for $\sin X$, $\cos X$, and $\tan X$; consult your language's documentation for full details. Unless otherwise stated in a problem description, it is *strongly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

Internet Access

Code Quest began allowing internet access during the competition in 2021; this was largely done out of necessity, as our competition had moved to a virtual setting in response to the global COVID-19 pandemic. However, we have made the decision to continue to allow internet access during our competitions, for both virtual and in-person locations.

Please note that if you are at an in-person location, you will be connecting to Lockheed Martin's guest network, and certain websites may be blocked. The Code Quest team does not have the ability to unblock any websites during the competition; if you require access to a blocked website, please contact the judging team for assistance. The judging team can also assist you with connecting to this network in the first place. You should have received a username and password to use when connecting to this network as part of your registration materials when you arrived; these will differ from the username and password you will need to log into the contest website.

While you will have the ability to access the internet during the contest, we ask and require that you do so responsibly. Please remember that Code Quest is a competition meant to test your team's programming skills, not those of random strangers on the internet. However, we understand that referring to documentation is a necessary part of software development. During the contest, we encourage you to make use of your programming language's official documentation, or basic tutorial websites that explain how to use key features of your programming language in general terms (such as W3Schools or official tutorials provided by Oracle, Microsoft, etc.). Our volunteers use these websites as well when we test the problems we present to you.

Please note that accessing these websites is against Code Quest rules, and accessing them may result in disqualification:

- Stack Overflow or similar websites that allow users to ask and receive answers to specific programming questions
- Gitlab or similar websites that allow storing and sharing of code snippets – this includes access to your own accounts, as using pre-written digital code is also against Code Quest rules
- ChatGPT or other websites that provide the ability to generate solutions to problems (ChatGPT may be blocked if you are at an on-site location, but this also includes any means by which to access ChatGPT or similar services through other websites which are not blocked)

All code that you submit during the contest must be your team's original work, created after the start of the contest. While we will not actively search for instances of dishonest behavior, our judging teams will be monitoring the competition closely, and any clear instance of dishonest behavior will be addressed, by means up to and including disqualification from the competition.

Thank you for your understanding and cooperation with helping us to maintain a fair and entertaining event. Best of luck today!

Problem 1: Color Coding

Points: 5

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

Problem Background

You and your coworker have developed secret signals to communicate between each other so others don't know the information you are conveying. The two key words used in your communications are "red" and "blue". You are tired of constantly searching all of your messages for these signals so you decide to write a program to find out if a line of text contains either of these strings.

Problem Description

Your task is to write a program that searches a given line of text for the strings "red" or "blue" (each string contains at most one of those colors). If red is found, return "red"; if blue is found, return "blue"; otherwise return "no color found". These strings are case sensitive, so if "RED" or "bLuE" is found then the program should still return "no color found".

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 which should be analyzed to see if "red" or "blue" are present:

```
5
This is a line of text that contains no colors
blue is my favorite color
I do not like the color red
This is a secret message, code red
A color that exists is BLUE!
```

Sample Output

For each test case, your program must print a single line containing either "red", "blue", or "no color found" (in lowercase letters):

```
no color found
blue
red
red
no color found
```

Problem 2: Number War

Points: 10

Author: Brett Reynolds, Bethesda, Maryland, United States

Problem Background

An elementary school teacher has developed a game for her students to help teach them about place values in larger numbers. However, with her large class, she needs a way to be able to keep an eye on everyone to ensure they're understanding the rules and playing correctly. She's enlisted your help to create a computer program that can review the cards students have and declare the winner of each round.

Problem Description

Based on the card game "War," the game "Number War" involves two players, who each start with a deck of playing cards with the face cards removed, leaving only the numbers 2 through 9.

In each round, each player secretly draws the top three cards from their deck. They must use these cards to create a two-digit number by placing one card into the "tens" position of the number and another into the "ones" place – the third card is set aside. For example, given a hand with the cards 4, 7, and 2, a student could make the two-digit numbers 24, 27, 42, 47, 72, or 74.

Whichever player ends up with the larger number keeps all six cards (their three, and their opponent's three). If the players create the same number, it's WAR! Each player draws three more cards, and they try again, with the winner taking all the revealed cards. The game continues until one player has fewer than three cards remaining.

The teacher has set up a camera above each student's desk, and by using an OCR scanner, she's able to provide your program with the numbers on the three cards each student draws each round. She'd like your program to determine who the winner of each round should be, assuming they're playing correctly. She'll then use that report to follow up with students who need a bit more help.

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 six integers separated by spaces, each ranging from 2 to 9 inclusive. The first three integers represent the cards drawn by Player 1; the last three represent those cards drawn by Player 2. Numbers may be repeated, even within a single player's hand.

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

Sample Output

For each test case, your program must print a single line declaring the result of each round, as follows:

- Print "PLAYER 1" if Player 1 is able to produce a larger two-digit number
- Print "PLAYER 2" if Player 2 is able to produce a larger two-digit number
- Print "WAR!" if the highest numbers both players can produce are equal

```
PLAYER 2
PLAYER 1
WAR!
```

Problem 3: Flex Time

Points: 15

Author: David Van Brackle, Marietta, Georgia, United States

Problem Background

At Lockheed Martin, employees often work on very large and very important projects, ranging from fighter planes to satellites. In contrast, researchers at the Advanced Technology Labs (ATL) frequently have tasks with short and aggressive timelines, and it's not unusual for an ATL'er to work on multiple projects in the same week or even the same day! Because of this, it's absolutely vital to keep an accurate accounting of employees' working hours.

Lockheed Martin uses a timecard application to track time, with a standard 40-hour work week. Within the timecard, employees can input the hours they have worked on separate projects throughout the day. If employees work more than 40 hours in a given week, they can be paid overtime or bank the excess. If employees work less, they must make up the difference.

Problem Description

Your task is to write a program which, given the employee's weekly timecards for each project, determines the hours they have worked over or under the standard 40 hours a week. If the employee worked more than 40 hours, print the difference as positive. If less than 40, print the difference as negative.

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 positive integer **N** indicating the number of projects the employee has worked on
- **N** lines, each with exactly seven positive integers separated by spaces representing the number of hours the employee has worked on each day of the week for a given project

```
3
3
10 10 0 0 0 0 1
0 0 5 5 0 0 0
0 0 4 0 4 0 0
1
10 10 10 10 0 0 0
4
0 0 12 0 0 0 0
7 7 0 10 0 0 0
3 3 0 2 0 0 0
0 1 0 0 0 0 0
```

Sample Output

For each test case, your program must print a line containing a single integer indicating the difference (positive or negative) between the hours worked and the expected goal of 40 hours.

```
-1
0
5
```

Problem 4: Lost in Translation

Points: 20

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

Problem Background

When working with embedded control systems, it's often necessary to use hexadecimal notation to send command strings to the device to instruct it to perform a given action. This helps reduce the complexity of the validation code needed to ensure the safe operation of the device.

Problem Description

You'll be provided with a series of strings containing hexadecimal values that represent command signals sent to an embedded device. You'll need to translate these values back into recognizable characters, then check to see if all of those characters are valid.

For example, a string such as...

```
Reset motor control
```

...might have been encoded into hexadecimal as...

```
52 65 73 65 74 20 6D 6F 74 6F 72 20 63 6F 6E 74 72 6F 6C
```

Each character is encoded using its ASCII value; 'R', for example, has a decimal value of 82 in ASCII. When converted to hexadecimal, it is equal to 52.

Once you decode the message back to its original form, you'll need to check to ensure all of the characters are recognizable by the embedded system. The valid characters include:

- All letters (uppercase and lowercase)
- All numbers (0-9)
- Spaces
- The following punctuation marks:
    - Period (.)
    - Comma (,)
    - Square brackets ([and])
    - Colons (:)

The hexadecimal strings may include encoded invalid characters; this may occur if the data is corrupted or was sent by a malicious attacker. Such strings must be identified so we can protect the embedded systems.

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 series of pairs of uppercase hexadecimal digits, with each pair separated by a space. These values represent the encoded command strings.

```
2
52 65 73 65 74 20 6D 6F 74 6F 72 20 63 6F 6E 74 72 6F 6C
43 4F 44 45 20 71 75 65 24 74 21
```

Sample Output

For each test case, your program must print a single line with the word 'VALID' if all characters in the provided string are valid, or the word 'INVALID' otherwise.

```
VALID
INVALID
```

Problem 5: Keeping Cool

Points: 20

Author: Lester Millan, Aguadilla, Puerto Rico, United States

Problem Background

Despite the amount of power they can produce, engines can be extremely delicate things. If they're not running in ideal conditions, they can gradually damage themselves to the point that they become completely useless. Care has to be taken to ensure that an engine is well-maintained and running smoothly to avoid disastrous results.

Problem Description

Lockheed Martin is developing a new temperature sensor to be installed in its fighter aircraft. This sensor will monitor the temperature of the aircraft's engine to ensure it's running at an ideal temperature. The sensor should set off warning alarms in any of the following situations:

- If any reading from the sensor is below a minimum temperature threshold; this means the engine isn't running as efficiently as it should, which may indicate damage
- If any reading from the sensor is above a maximum temperature threshold; this means the engine is running too hot and could become damaged quickly
- If the average of a series of readings from the sensor is above a warning threshold; this means the engine is running hotter than it should and may be receiving more damage over time, possibly due to overdue maintenance

Write a program that can accept a series of readings from the temperature sensor and display the appropriate output according to the rules above.

Sample Input

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

- A line containing the following information, separated by spaces:
  - A positive integer indicating the number of readings received from the sensor, N
  - A positive decimal indicating the minimum temperature threshold, L
  - A positive decimal indicating the warning threshold, A
  - A positive decimal indicating the maximum temperature threshold, H
- A line containing N positive decimals separated by spaces, representing the sensor readings over a period of time

For this problem, L < A < H for all test cases.

```
4
5 100.0 175.0 200.0
99.0 112.0 140.0 155.0 170.0
5 100.0 175.0 200.0
170.0 175.0 180.0 185.0 190.0
5 100.0 175.0 200.0
150.0 165.0 220.0 170.0 170.0
5 100.0 175.0 200.0
140.0 145.0 150.0 155.0 160.0
```

Sample Output

For each test case, your program must print a line indicating the warning system's output, as follows:

- If any temperature in the list is less than the minimum temperature threshold L, print "TOO COOL"
- Otherwise, if any temperature in the list is higher than the maximum temperature threshold H, print "TOO HOT"
- Otherwise, if the average of all temperatures in the test case is higher than the warning threshold A, print "WARNING"
- Otherwise, print "OK"

```
TOO COOL
WARNING
TOO HOT
OK
```

Problem 6: Deep Space Messages

Points: 25

Author: Andrew Aleman, Denver, Colorado, United States

Problem Background

While you and your team of astronauts are in orbit around Earth, one of the space station's communication arrays receives a series of messages. This isn't unusual... but what is unusual is that the messages didn't come from Earth! Your team needs to quickly decipher the messages to report them back to command.

Problem Description

You're able to determine that the messages contain a jumbled mass of letters, but occasionally contain a few numbers. The letters appear to be mostly background interference, but the numbers seem to be relevant. To decipher the message, you assign each number to a letter based on its position in the English alphabet (e.g. 1 = A, 2 = B, etc., up to 26 = Z).

Write a program that will translate numbers within each message string to their corresponding letters, then print the resulting text. You may assume that if two numbers are next to each other that they are part of the same integer.

Sample Input

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

```
4
dksieidik25nckdso15jklalfue21
aksleodnc1xsoeidmc18oeod5
utapszm14peu15owauet20
spepdoclqmen1oeudle12peeo15ae14epep5
```

Sample Output

For each test case, your program must print a single line containing the text deciphered from the received message using the process outlined above. Print text in lowercase letters.

```
you
are
not
alone
```

Problem 7: Mainframe Word Length

Points: 25

Author: Michael Warner, Denver, Colorado, United States

Problem Background

In computing, a "word" is the natural unit of data used in a processor design. It is a fixed number of bits (a "bit" is a single binary digit, 0 or 1). The bit length of registers, memory addresses, unsigned integers, instructions, and other quantities are typically the word length because it simplifies the computer design to keep these all consistent.

Modern computer architectures have settled on a *de facto* industry standard of 8-bit bytes, and a multiple of byte length for word length: usually 32-bit or 64-bit words. But earlier in computer history, the 8-bit byte hadn't been adopted, so there wasn't a "standard" word length. For the mainframes and minicomputers developed from the late 1940's to the mid-1970's, manufacturers chose word sizes that would look odd to us today.

Your team has been asked to help port an application from a modern 64-bit computer to run on some older architectures. You will need to see if the input values you're working with will fit within the target system's words.

Problem Description

Your program will be provided a word length (in number of bits) followed by a set of unsigned integer values representing input data. An "unsigned integer" is an integer value that doesn't use one of its bits to indicate if it's positive or negative; as a result, they are never negative. Your program should print the maximum unsigned integer value that can be represented in the word length, followed by TRUE if every input data value will fit in the provided word length. Word lengths can be any value from 4 to 64, inclusive. The input data values will always fit within 64 bits.

You may wish to check your language's documentation to determine the appropriate data type for containing unsigned 64-bit integers.

Sample Input

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

- A line containing a positive integer between 4 and 64 inclusive, representing the number of bits in a word in the legacy system architecture.

- A line containing between two and ten (inclusive) non-negative integers, separated by spaces and ranging from 0 to 18,446,744,073,709,551,615 inclusive, representing the input values to validate against the legacy architecture's word length.

```
2
12
1024 13 67 2999
36
3456 9999999 72165238905 5
```

Sample Output

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

- An integer representing the maximum unsigned integer value (in base-10) allowed by the legacy architecture's word length
- The word "TRUE" if every provided input value is less than or equal to the maximum unsigned integer value, or the word "FALSE" if at least one input value is greater than the maximum unsigned integer value, indicating it will not fit within the given word length.

```
4095 TRUE
68719476735 FALSE
```

Problem 8: AI Robot

Points: 30

Author: May Gu, Portland, Oregon, United States

Problem Background

Lockheed Martin is working on developing a reconnaissance robot that will eventually receive orders from an AI system. The AI system is still under development, but the robot is nearly ready to go; the team just needs to test that it's able to receive and process navigational commands correctly. They'd like you to write a program to simulate the orders that will be sent from the AI, then indicate the robot's expected position after it's followed those orders.

Problem Description

For the testing process, the robot will be placed on an X,Y coordinate grid, which can be considered to be infinite in all directions. To start with, you will be given the robot's current position and the direction it's facing:

- North (in the +Y direction)
- East (in the +X direction)
- South (in the -Y direction)
- West (in the -X direction)

The robot can receive three possible navigational commands:

- R – Turn right 90 degrees (e.g. North to East)
- L – Turn left 90 degrees (e.g. North to West)
- A – Advance one position on the grid in the direction the robot is facing

Commands are issued as a string of those characters and executed in sequence; the string "RAAL" tells the robot to turn right, move forward twice, then turn left. Given the starting position and facing and a series of instructions, you'll need to determine where the robot should end up, and in which direction it is facing.

Sample Input

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

- An integer representing the robot's starting X coordinate
- An integer representing the robot's starting Y coordinate
- An uppercase letter N, E, S, or W, representing the direction in which the robot is initially facing (North, East, South, or West, respectively)

- A string of one or more uppercase letters R, A, and/or L, representing a series of commands for the robot to follow.

```
2
0 0 N RAAL
7 3 N RAALAL
```

Sample Output

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

- An integer representing the robot's final X coordinate
- An integer representing the robot's final Y coordinate
- An uppercase letter N, E, S, or W, representing the direction in which the robot is facing after executing all instructions (North, East, South, or West, respectively)

```
2 0 N
9 4 W
```

Problem 9: Prime Phone Numbers

Points: 35

Author: Tyler Albaugh, Denver, Colorado, United States

Problem Background

You have always been a big fan of prime numbers and their unique properties. Recently, you came up with a fun challenge: take a list of phone numbers, split each one into three parts, and check if the numbers are pairwise coprime.

Problem Description

A standard US phone number can be thought of as a triplet of two three-digit numbers and one four-digit number - for example, (800) 634-5789, where a=800, b=634, c=5789. You will be given a series of phone numbers and your task is to check if the three numbers, a, b, and c, are pairwise coprime.

Two numbers are coprime, or relatively prime, if they share no common factors besides 1. A triple (three numbers a, b, c) is pairwise coprime if each of the pairs (a,b), (b,c), and (a,c) are coprime. For example, the set (8, 9, 11) is pairwise coprime, while (8, 9, 10) is not (8 and 10 share a common factor, 2).

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include a single line containing a phone number in (XXX)YYY-ZZZZ format.

```
4
(123)456-7890
(121)122-1235
(345)678-9012
(457)124-2255
```

Sample Output

For each test case, your program must print a single line containing the word "TRUE" if the set of numbers represented by the phone number is pairwise coprime, or "FALSE" otherwise.

```
FALSE
TRUE
FALSE
TRUE
```

Problem 10: Test Coverage

Points: 40

Author: Michael Warner, Denver, Colorado, United States

Problem Background

At Lockheed Martin, developing a good software product doesn't just require writing good code; it also needs good design and good testing. A group of people may all have various ideas about what some piece of software should do and how it should work. Once they come to agreement on those details, they eventually get documented as requirements in a formal document called a Software Requirements Specification. Each requirement is carefully written to avoid any confusion and receives a unique identifier to make it easier to reference.

Before a system gets delivered – regardless of whether it's a calculator program or the software that drives a massive rocket's launch system - every subsystem needs to be thoroughly tested to ensure that it meets the requirements. Otherwise, you might wind up with a giant explosion or merely a billion-dollar paperweight. To ensure that every requirement gets tested and there is nobody saying "oops, must have forgot about that one" 2 minutes after launch, the test procedures and their associated requirements are all cross-referenced in a big spreadsheet called the Requirements Traceability Matrix.

Problem Description

Requirements for a launch computer on a rocket may look something like this:

- LC-001: The launch computer shall stop the countdown if it has not received GO_FOR_LAUNCH status from the control tower when the countdown time reaches T-00:00:10.
- LC-002: The launch computer shall initiate the launch sequence at T-00:00:00 if it has received GO_FOR_LAUNCH status from the control tower and its internal diagnostic status indicates no errors.
- LC-003: When the launch sequence has been initiated, the launch computer shall command the engines to ignite.

...And so on, and so on. As you can see, each requirement has a unique identifier, made up of a prefix and a number. In this example, those identifiers are LC-001, LC-002, LC-003. As noted above, each of these requirements must be tested using a defined procedure. These test procedures each have their own identifiers: LCTP-001, LCTP-002, LCTP-003, and so on.

The Requirements Traceability Matrix indicates which test procedure(s) test which requirement(s). A single test procedure might test multiple requirements. On the other hand, there might be multiple

test procedures needed to test a single requirement. If a requirement number matches a test procedure identifier, it is only a coincidence.

The Requirements Traceability Matrix for multiple subsystems will be given as input. Your program must read in the list of requirement identifiers and the list of requirements covered by each test procedure for the subsystem and ensure that every requirement is covered by at least one test procedure. If any requirements are not covered by a test procedure, your program must print out those requirement identifiers. If every requirement is covered by a test procedure, your program must print out "FULL COVERAGE".

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, N, indicating the number of entries in the Requirements Traceability Matrix
- A line containing a list of requirement identifiers separated by commas. Each requirement identifier will contain a prefix of letters and/or dashes, followed by a single integer which may include leading zeroes. These numbers may be presented in any order and may not represent a contiguous set of numbers (requirements get added and removed all the time!).
- N lines each containing an entry in the Requirements Traceability Matrix. Each entry will include the following information, separated by commas:
    - A test identifier, which will contain a prefix of letters and/or dashes, followed by a single integer which may include leading zeroes. The prefix will not match that of any requirement in the same test case. Numbers may be presented in any order and may not represent a contiguous set of numbers.
    - One or more requirement identifiers from the list previously given.

```
2
1
LC-001,LC-003,LC-002,LC-015,LC-105
LCTP-002,LC-003,LC-001
3
MC-515,MC-616,MC-007,MC-009,MC-112
MCTP-001,MC-007,MC-112
MCTP-005,MC-515
MCTP-098,MC-616,MC-009
```

Sample Output

For each test case, your program must print the requirement identifiers that are not covered by a test procedure, separated by commas. Print requirement identifiers in default string ordering (that is, sorted according to the ordering given by the US ASCII table in the reference materials). If all requirements are covered by a test procedure, your program must print FULL COVERAGE.

```
LC-002,LC-015,LC-105
FULL COVERAGE
```

Problem 11: Cipher Clash

Points: 45

Author: Hamzah Abdulrazzaq, Mooretown, New Jersey, United States

Problem Background

In a Lockheed Martin satellite mission, secure communication between ground control and the satellite relies on a clever encoding method: anagrams. An anagram is a word or phrase formed by rearranging the letters of another word or phrase, using all of the original letters exactly once. This encoding ensures that even if a message is intercepted, it is hard to decode without advanced techniques. Your mission is to help verify if the communication between ground control and the satellite is secure by checking for these anagrams.

Problem Description

Given two sentences, one from ground control and one from the satellite, determine if a specific pair of words (identified by their positions in the sentences) are anagrams. For each test case, you'll be given two integers representing the positions of a word in each sentence to compare (the first word in each sentence is represented by 1, the second by 2, and so on). Extract the corresponding word from each sentence and check to see if they are anagrams.

If the words are anagrams, output "Verified;" otherwise, output "Intercepted."

Remember, anagrams are words that use the same letters in different orders. These pairs of words are all anagrams:

- "secure" and "rescue"
- "signal" and "aligns"
- "listen" and "silent"
- "earth" and "heart"

In each test case, the words identified by the position numbers will be different from each other; you will not be asked to compare a word to itself.

Sample Input

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

- A pair of index numbers, presented in `[X,Y]` format, representing the positions of the words in each sentence to compare. X indicates the position of the word in the first sentence; Y indicates the position of the word in the second sentence. For both values, 1 represents the first word, 2 represents the second word, etc.

- The first sentence to evaluate. The sentence will be wrapped in double quotes ("), and may contain any combination of characters other than double quotes.
- The second sentence to evaluate, in the same format as the first sentence.

```
2
[6,8] "The missile defense system requires secure protocols" "Specific rules must be followed during the rescue mission"
[6,5] "The satellite must be launched into orbit tonight" "Their mission includes monitoring the robot movement"
```

Note that the sample input only contains three lines of text; the two test cases are wrapping to new lines due to their length.

Sample Output

For each test case, your program must print a single line of text containing the word "Verified" if the identified words are anagrams of each other, or the word "Intercepted" otherwise.

```
Verified
Intercepted
```

Problem 12: Coloring the Planet

Points: 45

Author: Gary Hoffmann, Denver, Colorado, United States

Problem Background

You have joined a team developing a digital twin of the Earth for the National Oceanic and Atmospheric Administration. This new system will help climate scientists better understand the changing climate. Your team selects sea surface temperature as the first variable to show and you've been assigned the job of converting the temperature values into colors so that the scientists can visually inspect the temperature data coming out of the digital twin.

The colors will be selected by a color bar (gradient) based upon the sea surface temperature in kelvin.

Problem Description

Please note that this problem overrules the general guidelines for rounding numbers provided in the Reference Materials.

Your program will be given the color bar as a list of control points (position on the bar and the color they represent). The color bar will be provided as a row containing the number of control points and the start, end temperature of the color bar (in Kelvin). That will be followed by a single row for each control point which will have the position of the point (0 = start of the bar, 1 = the end of the bar) followed by a red, green, and blue color values (from 0 to 255) for that control point.

The following is an example color bar with sample input:

```
5 273 315
0 0 0 255
0.25 0 255 255
0.5 0 255 0
0.75 255 255 0
1 255 0 0
```

Your program must read in a sea surface temperature and use a linear interpolation between the control points to determine the correct color for that temperature and print a single line containing the red, green, and blue components of the color. Because the linear interpolation will return a floating-point number, use a floor function to strip off the decimal portion of the red, green and blue values.

A linear interpolation can be used to estimate a value in between two known points (our control points in this problem). The formula for a linear interpolation is (where a and b are the points and t ranges from 0 to 1):

$$lerp(a, b, t) = (1 - t) * a + t * b$$

For example:

$$lerp(5, 10, 0.0) = 5$$

$$lerp(5, 10, 0.5) = 7.5$$

$$lerp(5, 10, 1.0) = 10$$

In order to align your calculations with RGB color values, you'll need to **round your results down** if they contain decimals.

The formula for finding the position of a value relative to the minimum and maximum endpoints will need to be used in your calculations.

$$\frac{(value - min)}{(max - min)}$$

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 integers separated by spaces, representing a color bar definition. These integers represent:
  - N, the number of control points
  - L, the low temperature in kelvin
  - H, the high temperature in kelvin
- N lines representing control points, each containing the following values separated by spaces:
  - A decimal number between 0 and 1 inclusive representing the position of the control point along the line. Control points will be listed in ascending order by this position.
  - An integer between 0 and 255 inclusive representing the red value of that control point
  - An integer between 0 and 255 inclusive representing the green value of that control point
  - An integer between 0 and 255 inclusive representing the blue value of that control point
- A line including an integer, between L and H inclusive, representing the temperature in kelvin that needs to be converted into a color.

Due to its length, the sample input is provided on the next page to avoid breaking across pages.

```
2
5 273 315
0 0 0 255
0.25 0 255 255
0.5 0 255 0
0.75 255 255 0
1 255 0 0
300
2 250 350
0 0 0 255
1 255 0 0
310
```

Sample Output

For each test case, your program must print the red, green, and blue values as integers **(use the floor function to ensure these are integers)** separated by a space, representing the color for that temperature.

```
145 255 0
153 0 102
```

Problem 13: Timecard Helper

Points: 50

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

Problem Background

At Lockheed Martin, we take great pride in doing what's right. A large part of that involves accurately accounting for the time we spend working each day so that we can accurately bill our customers. As part of an agile development team, you'll most likely have multiple tasks assigned to you on any given day. On top of that, you may also have to help out other members of your team with what they're working on. Keeping track of how much time you spend on each task can get a little tricky when you bounce back and forth between different efforts all day.

Problem Description

For this problem, you've been tasked with making an application to help employees keep track of the hours they've worked on given tasks and summarize that information in a format they can easily enter into their timecards at the end of each day.

You will be provided with a list of task names and their associated charge codes, a 7-digit number that is entered on their time cards. The next input will be a list of task descriptions and how long an employee spent on each task. Sometimes, employees bounce back and forth between tasks, so you may see the same task multiple times. Also, some tasks may use the same charge number.

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:
  - T, representing the number of tasks an employee might be working on, and
  - E, representing the number of times an employee actually worked on a task
- T lines, each containing the name of a task (containing upper- and lower-case letters and spaces), a colon (:), and the task's associated charge number (a seven-digit number). Each task will appear in this list at most one time.
- E lines, each containing the name of a task (from the list provided previously), a colon (:), and a positive number representing the number of hours spent working on that task. A task may appear in this list multiple times.

```
3
6 5
Code App:5944035
Write Documentation:9556458
Attend Design Review:7653329
Deploy To Production:5499311
Attend Planning:6214339
Write Test Procedure:3270799
Code App:2.0
Attend Planning:2.0
Deploy To Production:4.5
Attend Planning:2.6
Write Documentation:5.7
3 10
Write Test Procedure:9543441
Attend Planning:9066150
Write Unit Tests:9589130
Attend Planning:3.7
Attend Planning:0.6
Write Unit Tests:5.9
Write Unit Tests:4.9
Write Test Procedure:1.7
Attend Planning:5.6
Write Unit Tests:0.9
Write Unit Tests:2.8
Write Test Procedure:4.0
Attend Planning:5.0
3 4
Deploy To Production:8871036
Attend Planning:2787693
Attend Design Review:4174936
Attend Design Review:3.3
Deploy To Production:4.9
Attend Design Review:1.6
Deploy To Production:1.1
```

Sample Output

For each test case, your program must print output reflecting what the employee should enter into their timecard.

If the total time worked does not exceed 24 hours, print one line for each of the employee's charge numbers, in increasing numeric order. Each line must contain the charge number, a colon (:), and the total time worked on tasks using that charge number. Print a single decimal place for the hours worked on each charge number.

However, if the total number of hours worked by the employee exceeds the number of hours in a day, print a single line containing the word "Error". In a case where an error occurs, do not print any lines of charge hours.

```
5499311:4.5
5944035:2.0
6214339:4.6
9556458:5.7
Error
4174936:4.9
8871036:6.0
```

Problem 14: Going in Circles

Points: 55

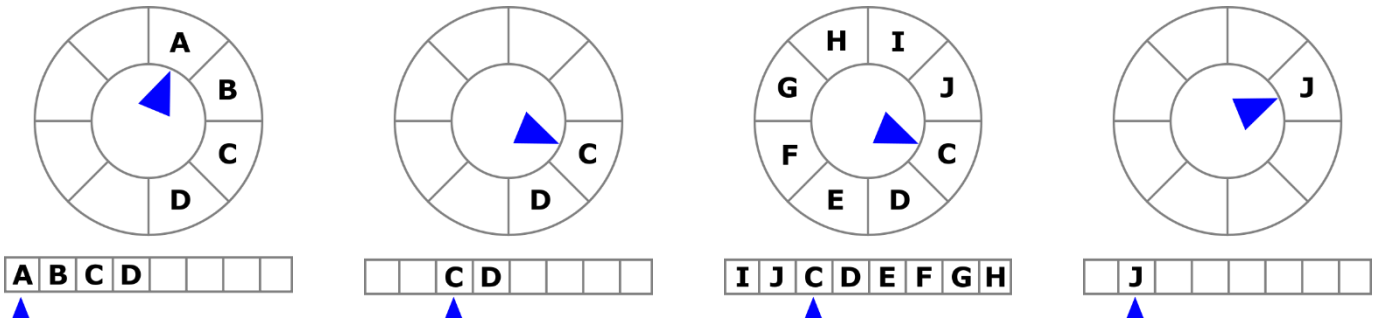
Author: Jonathan Tran, Grand Prairie, Texas, United States

Problem Background

When you're watching a video on YouTube or some other website, your browser won't download the entire video at once; it'll only download a few seconds ahead of where you currently are. This process, called "buffering," allows the video to continue playing for a short period, even if your internet connection is briefly disrupted. Video buffering frequently makes use of a data structure called a "circular buffer."

Circular buffers have a fixed size and store data in a circular (wrap-around) pattern. These are often used where memory is limited, and where dynamically allocating memory (increasing memory usage as needed) isn't ideal. Circular buffers work by filling a pre-allocated array with data as it arrives. A memory pointer keeps track of the "start" of the buffer; even though a circle doesn't have a start point, the computer has to know where to begin reading data in the buffer when needed. As data is read from the buffer, it is removed, freeing up space for more data.

See the example below; in this diagram, the characters A B C D are stored into a circular buffer with a size of 8. Two of these are then read and removed: A and B. The "start" pointer is then moved ahead to the next unread character, C. Next, the characters E F G H I J are stored; I and J wrap around to get placed in A and B's now-vacant positions. Seven characters are then read; the pointer marking the "start" of the buffer also wraps around when it reaches the end of the array.



Problem Description

Your team is working on implementing a circular buffer for use with a VR flight simulator being built with Lockheed Martin's Rotary and Mission Systems. The quality control team wants some tests built to demonstrate that the buffer algorithm works as intended. You need to create a program that can accept the following commands (which will be used in a script provided by the quality control team) and properly process them:

- ADD - This command requires your program to add one or more characters to the buffer, starting at the first empty position after the read position. If the buffer is or becomes full, existing data should be overwritten in a circular manner, and the read position should be advanced to the first position that was not overwritten.
- CONSUME - This command requires your program to remove the specified number of characters from the buffer, starting with the character at the current read position. The read position should be advanced to the position after the last character removed. If the number of characters to consume exceeds the number actually stored in the buffer, all characters are consumed and the buffer will be emptied.
- SHOW - This command requires your program to print some information about the current state of the buffer. If the buffer contains an odd number of characters, your program should print the character located at the midpoint between the current read position and the last character within the buffer. If the buffer contains an even number of characters, your program should print the two characters to either side of that midpoint. If the buffer is empty, your program should print "Empty". This command does not add or remove data from the buffer, and does not change the read position.

Sample Input

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

- A line containing two positive integers separated by a space:
  - N, representing the number of commands included in this test case
  - S, representing the size of the circular buffer.
- N lines each containing a single command and its arguments. The command and each argument (if any) will be separated by spaces. Possible commands and arguments include:
  - ADD, which will be followed by one or more visible and printable ASCII characters (see the US ASCII table in the reference information for examples; spaces will not be given as arguments)
  - CONSUME, which will be followed by a single non-negative integer
  - SHOW, which has no arguments

```
2
4 5
ADD A B C
SHOW
ADD D
SHOW
9 6
ADD a b c d
CONSUME 1
SHOW
CONSUME 4
SHOW
ADD e f g h i
SHOW
ADD j k l
SHOW
```

Sample Output

For each test case, your program must print the output from any SHOW commands received during the test case. The result of each command should be printed on a separate line; when two characters must be printed, print them in the order read and separated by a single space.

```
B
B C
c
Empty
g
i j
```

Problem 15: Workflow Tracking

Points: 60

Author: Michael Wells, Grand Prairie, Texas, United States

Problem Background

Managing software is a difficult process; in large projects, it can be difficult to add new features or update existing ones without introducing new bugs or otherwise breaking other features. Simply adding new features without considering the impacts can lead to significantly higher costs and delivery times. Lockheed Martin often uses a process managed by a "Change Review Board" to ensure requests for new features are fully understood and evaluated.

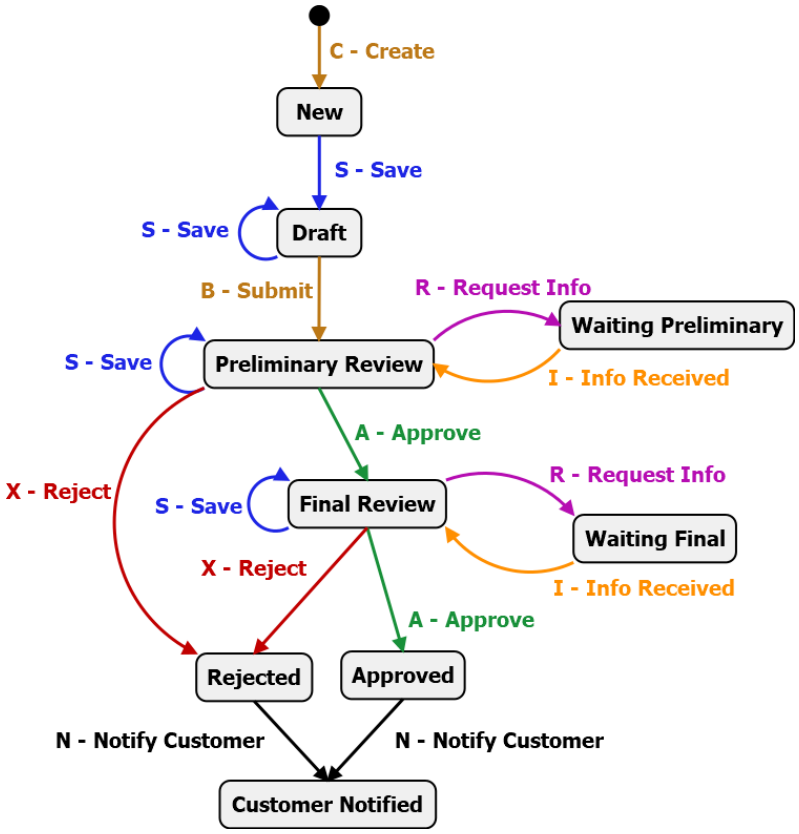
You've been appointed as the lead for your project's Change Review Board, and would like to streamline things by creating an automated process to handle requests from the customer.

Problem Description

Your project's Change Review Board generally follows the process shown in the workflow diagram at right.

A change request goes through a series of states (from top to bottom) as it's reviewed by the Change Review Board. Each of the boxes in the diagram shows a state that a request could be in at any given time; each arrow shows how the request can move from one state to another, and why. You'll need to create a program that implements this workflow and can validate if a request is being processed correctly.

For example, if a request is in the "Final Review" state, a reviewer could "Approve" or "Reject" it; they cannot "Submit" the request, as there's no arrow coming from "Final Review" with that label.



The rest of your team is working on creating an interface for this tracking tool; they'll develop a menu that provides controls for each of the actions (arrows) in the diagram above. These will be identified in the software using single-character codes, as follows:

- C - Create

- S - Save
- B - Submit
- R - Request Info
- I - Info Received
- X - Reject
- A - Approve
- N - Notify Customer

Your task is to work on creating a program that can validate that these actions are being executed correctly. You'll be given a string showing a list of command codes. You must output what state the request is in following each command. If a given command is invalid for the request's current state, don't change the request's state, but instead output "Invalid Command".

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 information, separated by spaces:

- A positive integer, representing the request's tracking ID
- Two or more uppercase letters, from the list provided above, representing commands that were attempted in relation to this request, in the order they were attempted.

```
4
1 C S B R I A X N
2 C B S B A A N
3 S C S B X N A
4 C S B A R I A N
```

Sample Output

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

- The tracking ID for the request
- A space
- For each command issued for the request:
    - If the command was not the first issued for the request, a 'greater-than' sign (>)
    - The name of the state resulting from execution of the command (as provided in the diagram above), or "Invalid Command" if the given command is not valid for the current state of the request

Note that due to the length of each line of output, the lines in the sample provided below are breaking across multiple lines within this document. Please download the sample output file from the contest website for a more accurate depiction of the output. Each test case's output consists of a single line, starting with an integer.

```
1 New>Draft>Preliminary Review>Waiting Preliminary>Preliminary Review>Final
Review>Rejected>Customer Notified
2 New>Invalid Command>Draft>Preliminary Review>Final Review>Approved>Customer
Notified
3 Invalid Command>New>Draft>Preliminary Review>Rejected>Customer Notified>Invalid
Command
4 New>Draft>Preliminary Review>Final Review>Waiting Final>Final
Review>Approved>Customer Notified
```

Problem 16: Reciprocal Repetition

Points: 65

Author: Kelly Reust, Denver, Colorado, United States

Problem Background

Any time you calculate the reciprocal of a number (divide 1 by that number), you're going to get a decimal value. What's interesting about that decimal value, is that it will eventually start to repeat itself. For example, if you divide 1 by 3, you get 0.333333333…. Those 3's will never stop repeating themselves. Sometimes this repetition can include multiple numbers; dividing 1 by 7 results in 0.142857142857142857…. Here, the repeating group is 142857. Another way to write this number is $0.\overline{142857}$; the bar over those numbers indicates that they repeat indefinitely. Even if a number doesn't appear to repeat indefinitely, as with 1 / 10 = 0.1, it still has a repeating group of 0; $0.1\overline{0}$. We just typically ignore the extra zeroes.

Problem Description

Let's write a program to calculate the repeating portion of several reciprocal numbers. Given an integer value, you will need to calculate the reciprocal of that number, then identify the repeating group of that reciprocal as shown above.

WARNING: Computers are notoriously bad about storing decimal numbers in their memory. Computers store numbers in a binary format, and this can cause inaccuracies when working with decimal numbers. We strongly recommend that you do not rely on simple division to create your reciprocals. Using long division will be more likely to yield accurate numbers and avoid wrong answers.

$$
\begin{array}{rl}
 & 0.\ 2\ \ 5\ \ \overline{0} \\
4\,)\overline{1.\ \ 0\ \ 0\ \ 0} \\
- & 8 \\
 & 2\ \ 0 \\
- & 2\ \ 0 \\
 & \ \ \ 0 \\
 & - \ 0 \\
 & \ \ \ \ \ 0
\end{array}
$$

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 between 2 and 100 inclusive.

```
3
3
7
10
```

Sample Output

For each test case, your program must print a single line containing the repeating group of the reciprocal of the given number.

```
3
142857
0
```

Problem 17: Permissions Check

Points: 70

Author: Adrienne McKee, Huntsville, Alabama, United States

Problem Background

File permissions are an essential piece to the security model used by Linux systems. They determine who can access files and folders (or directories) on a system and how. On the Linux system files and folders the owner, group, and others are granted access based on the permission. We want to make sure that the permissions follow the Least Privilege principle, meaning that everyone has the access they need, but nothing more than that.

As part of your internship with Lockheed Martin's Corporate Information Security division, you have been tasked with writing a program to determine if all of the files or directories in a given folder contain the correct permissions, meaning they follow the Least Privilege principle. The input to your program will be a cleaned-up listing of a directory using the `ls -lAR` command which will recursively find all files or directories and print out their permissions.

Problem Description

When the command `ls -l` is executed from a Linux command line, the output is a group of metadata that includes the permissions on each file.

```
-rw-rw-r--. 1 user group 10 Feb 26 07:08 file1
-rw-rw-r--. 1 user group 10 Feb 26 2017 file2
```

Here are the components of the listing, separated by spaces:

- File Attributes:
  - File type: 1 character describing the type: - (file), or d (directory)
  - Permission settings: 9 characters in groups of three; each group contains "rwx" in that order, although each letter can be replaced with a dash (-). "r" stands for read, "w" for write, and "x" for execute; a "-" means that permission is denied.
    - Characters 1-3 are the owner's permissions.
    - Characters 4-6 are the group's permissions.
    - Characters 7-9 are the others' (not the owner or member of the group) permissions.
  - Extended attributes: For this example, will always be a dot (.)
- Number of Hard Links
- User owner: user
- Group owner: group
- File size, usually in MB
- The date and time at which the file was created/modified

- Name of the file

You'll need to inspect each file and directory you've been asked to check and compare their permissions to what we want set. The permissions must match exactly; if a user or group has either more or fewer permissions than they should, it's incorrect and must be fixed.

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 values separated by spaces:

A single line containing the following information, separated by spaces:

- An uppercase letter, indicating whether to evaluate only Files (F) or Directories (D).
- Three characters, representing the desired owner Permissions (a lowercase "r" or a dash, then a lowercase "w" or a dash, then a lowercase "x" or a dash).
- Three characters, representing the desired group Permissions (a lowercase "r" or a dash, then a lowercase "w" or a dash, then a lowercase "x" or a dash).
- Three characters, representing the desired other user Permissions (a lowercase "r" or a dash, then a lowercase "w" or a dash, then a lowercase "x" or a dash).
- A positive integer, N, indicating the number of lines of input that follow.

The next N lines contain the details for each directory you need to review. Each directory will be identified by its relative path, followed by a colon (:); all relative paths will start with "./". Following that will be one or more lines containing the `ls -l` output from that directory, as outlined above.

```
2
F r-- r-- r-- 5
./lookHere:
drwxrwxrwx. 1 user group 10 Feb 26 2017 directory1
-r-xr-xr--. 1 user group 10 Feb 26 2017 file1
./lookHere/directory1:
-r--r--r--. 1 user group 10 Feb 26 2017 file2
D r-- r-- r-- 5
./lookHere:
drwxrwxrwx. 1 user group 10 Feb 26 2017 directory1
-r-xr-xr--. 1 user group 10 Feb 26 2017 file1
./lookHere/directory1:
-r--r--r--. 1 user group 10 Feb 26 2017 file2
```

Sample Output

For each test case, and for each directory being searched in the order provided, your program must print out a report as follows:

- If the permissions for all files/directories in a directory are correct (or there are no files/directories to review), print a single line containing the relative path to the directory, a colon (:), a space, and the words ALL GOOD.
- If any permissions are incorrect, print a line containing the relative path to the directory, followed by a colon. Then print a separate line for each file with incorrect permissions, in the order they were originally listed, in the following format:
    - The name of the file
    - The text ": WRONG PERM: "
    - The user categories with incorrect permissions (one or more of Owner, Group, and/or Other, in that order) separated by a comma and a space.

In nested directory structures, only consider those immediate children of a directory when deciding to print "ALL GOOD" or not; a child directory with a file with bad permissions doesn't reflect against the parent directory.

```
./lookHere:
file1: WRONG PERM: Owner, Group
./lookHere/directory1: ALL GOOD
./lookHere:
directory1: WRONG PERM: Owner, Group, Other
./lookHere/directory1: ALL GOOD
```

Problem 18: Just One Thing

Points: 75

Author: Javier Jimenez, Marietta, Georgia, United States

Problem Background

Lockheed Martin has been contracted by the Transportation Security Administration to develop some new airport security equipment, to be trialed at Atlanta International Airport. They'd like to implement a process in which passengers will place only a single item in each bin to be passed through the x-ray scanner; this will help TSA agents better identify what's going through the scanner. To help enforce this, they're planning to set up a camera that will watch as bins approach the x-ray scanner. If a bin contains more than one item, it will flash a warning light so agents know to rearrange things.

Problem Description

You'll be provided with an image of a security bin in X PixMap, or XPM, format. XPM files are text-based image files that use ASCII characters to represent individual pixels in an image. Your task is to identify if there are multiple objects within the bin. Since the bin itself is a uniform gray color (specifically #C0C0C0), you should be able to identify objects by finding areas of the image that are *not* that color. Objects may be any shape and size, and may contain any combination of colors other than #C0C0C0.

XPM files are easily viewed within a text editor, and follow a consistent format:

- The first line contains a comment identifying the file as an XPM file (this is irrelevant to this problem):
  `/* XPM */`
- The second line declares a string array in the C programming language (also irrelevant for our purposes):
  `static char * file_name_here_xpm[] = {`
- The third line is the first one to contain any useful information; it contains a string with four integers, separated by spaces:
  - The width of the image, in pixels
  - The height of the image, in pixels
  - The number of colors defined in the image
  - The number of characters used to define each pixel (for this problem, this value will always be 1)
- The next several lines – the exact number determined by the third value in the previous string - will contain strings that define the colors used in the image. Each color is defined as follows:
  - An ASCII character (including spaces!) used to represent a pixel in the image
  - A tab character

    - A lowercase letter indicating the type of the color; for this problem, this will always be 'c', for "color"
    - A space
    - A hexadecimal RGB color string, such as #C0C0C0 for the bin's background.

The remaining lines contain strings representing the image itself. The number of strings will be equal to the image's height, and the length of each string equal to its width. Each character represents a single pixel; the color of that pixel is determined using the color mapping provided above.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include a single image in XPM format; while this format is generally outlined above, to review:

- The first two lines of each test case contain no relevant data and may be ignored
- The third line of each test case will contain a string (wrapped with double quotes, and followed by a comma) containing the following positive integer values separated by spaces:
    - W, the width of the image in pixels
    - H, the height of the image in pixels
    - C, the number of colors defined for potential use in the image
    - P, the number of characters representing each pixel. This value will always be 1.
- C lines containing color definitions. Not all colors defined here will necessarily be used within an image. These are also defined as strings (wrapped with double quotes, and followed by a comma), using the following format:
    - A single printable ASCII character, which is the character that will be used to represent pixels in this color. This character will not be a tab, double quote, or backslash, but may be any other ASCII character appearing in the table in the Reference Materials.
    - A tab (\t), a lowercase c, and a space
    - A hexadecimal color code, prefaced with a # character
- H lines, each containing a string of length W wrapped in double quotes and followed by a comma (except for the last line, which is followed by a closing curly bracket and semicolon). These strings represent the pixels in the image, and will contain characters listed in the color definition section above.

Due to the length of the sample input, it is not replicated here; please download the sample input from the contest website's "Problemset" tab.

Sample Output

For each test case, your program must print a single line containing the word "OK" if the image contains at most one object; or containing the word "WARNING" if the image depicts more than one object.

```
OK
WARNING
```

Problem 19: Batch It Up

Points: 80

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

Problem Background

At Lockheed Martin, many programs rely on being able to access data obtained from other systems. Some of these programs connect directly to the source databases, whereas others rely on manual export/import processes. Ensuring that your program can effectively "ingest" this external data is a key step towards developing a functional system. However, sometimes the amount of data you're trying to import can be a bit excessive. When that happens, it's important to break it down into smaller batches.

Problem Description

You're working with Lockheed Martin Aeronautics to develop a system to manage an inventory tracking system for parts for fighter aircraft. Your team has been provided with a series of XML files that contain information about the parts held at various Air Force bases, and you need to load this into your database. Unfortunately, there are some difficulties with this: your database can't read XML files directly, the format of the data is somewhat inconsistent, and the files are far larger than your database can handle at one time anyway. Your team needs to read the data from the XML files, process it into a legible format, and divide the data into smaller batches your database can handle.

Each part listed in the XML files has three main data points associated with it: its name, its serial number, and its part number. None of these data points are unique on their own, however the combination of part number and serial number will be unique for each part.

Within the XML files, parts may be listed in one of three formats. In each of the examples below, the part name is "Name", the part number is "PartNum", and the serial number is "SerialNum":

- An empty/self-closing element. Attributes may be presented in any order.
  ```
  <part name="Name" number="PartNum" serial="SerialNum" />
  ```
- A single element. The content of the element will be the part's serial number; the other attributes may be presented in any order.
  ```
  <part number="PartNum" name="Name">SerialNum</part>
  ```
- A set of nested elements, each listing one property of the part, and appearing in any order.
  ```
  <part>
  <name>Name</name>
  <serial>SerialNum</serial>
  <number>PartNum</number>
  </part>
  ```

Elements following the first two formats will be presented on a single line; the third format will be spread across five lines, as shown above.

Your solution will need to be able to process and store the information for these parts and report them in batches for addition to your database. Make sure that all records are 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 a positive integer, representing the number of parts to include in each batch.
- A line containing the opening parent <parts> tag, representing the start of an XML file containing parts data.
- One or more lines containing the content of the XML file. These lines will contain valid XML elements matching the formats described above. Lines will not be indented in any way. Part numbers, serial numbers, and part names may contain any alphanumeric character and/or spaces.
- A line containing the closing parent </parts> tag, representing the end of the XML file.

```
1
3
<parts>
<part name="Thing" serial="123" number="ABC" />
<part serial="456" number="DEF" name="Other Thing" />
<part number="ABC" name="Thing">456</part>
<part number="123" name="Doohickey">ABC</part>
<part>
<number>JKL1</number>
<serial>42</serial>
<name>Wrench</name>
</part>
<part>
<name>Spanner</name>
<number>JKL1</number>
<serial>999</serial>
</part>
</parts>
```

Sample Output

For each test case, your program must print the part data, in the order it was listed in the original XML file, in batches not exceeding the batch size provided in the input. Use this format for each batch containing one or more parts:

- A line containing an integer with the batch number (the first batch in each test case should be batch 1; this resets to 1 with every new test case)
- A line for each part within the batch, containing:
    - The part number

    - A comma (,)
    - The serial number
    - A comma (,)
    - The part name

```
1
ABC,123,Thing
DEF,456,Other Thing
ABC,456,Thing
2
123,ABC,Doohickey
JKL1,42,Wrench
JKL1,999,Spanner
```

Problem 20: CDRL Delivery Date

Points: 85

Author: Adrienne McKee, Huntsville, Alabama, United States

Problem Background

A Contract Data Requirements List (CDRL) is an item that is due to the customer by a certain date. Often these dates are a certain number of calendar days before or after the date of a certain event. Teams must know not only by which date the item is due to the customer, but because the end dates can fall on a weekend or holiday, the team must ensure the deliverable date is a valid workday.

In order to ensure that all documents are delivered on time, Lockheed Martin Enterprise Operations is developing a tool for planning out project deadlines, and your team has been asked to help.

Problem Description

Your program should calculate the date the CDRL is due, and if the date falls on a weekend (Saturday or Sunday) or holiday, the due date should be adjusted based on the following guidance. The CDRL will be due either before or after a designated date of an event. The date of the event and whether the CDRL is due before or after that event is given as input.

- If a CDRL due date is due after the event, and the due date falls on a weekend or holiday, you will need to adjust that date to be the closest non-holiday weekday after the previously calculated due date.
- If a CDRL due date is due before the event, and the due date falls on a weekend or holiday, you will need to adjust that date to the closest non-holiday weekday before the previously calculated due date.
- If a CDRL due date is calculated to be a holiday which also falls on a Saturday or Sunday, the holiday observance date will need to be treated as a holiday.
    - If a holiday falls on a Saturday, the holiday observance date will be the preceding Friday.
    - If a holiday falls on a Sunday, the holiday observance date will be the following Monday.

Lockheed Martin has provided a list of holidays which are considered undeliverable dates. Some holiday dates are consistent every year, while others are dependent on a specific day in a month which may change every year. Since your program will be used going forward, your program must account for these different dates for any year. These holiday dates will need to be calculated dynamically during your program's runtime. All the CDRLs are related to contracts that involve employees from multiple different countries. So, holidays celebrated in a variety of countries are included in the list. Below is the list of holidays that need to be evaluated during your program's CDRL due date calculation.

Holiday Name	Country	Occurrence

All Saints' Day	PL	November 1
ANZAC Day	AU, NZ	April 25
Canada Thanksgiving	CA	2nd Monday in October
Christmas Day	Multiple	December 25
Code Quest Academy Day	N/A	First Sunday in November
Commemoration Day of King's Father	CAM	October 15
Memorial Day	US	Last Monday in May
National Day	SG	August 9
New Year's Day	Multiple	January 1
Summer Bank Holiday	UK (Except Scotland)	Last Monday in August
US Thanksgiving	US	4th Thursday in November
US Thanksgiving Observed	US	The Friday after "US Thanksgiving"

Sample Input

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

- A line with three values, separated by a space. The three values are defined as follows:
  - The date of the event in format YYYY/MM/DD.
  - The word "BEFORE" or "AFTER" in all capital letters, which represents if the CDRL is due before or after the event date.
  - A positive integer, which represents the number of days before or after the event date that the CDRL is due. **10**

```
2028/03/25 AFTER 71
2025/10/03 BEFORE 104
2028/04/22 AFTER 87
2025/12/09 BEFORE 38
2030/09/26 BEFORE 48
2026/01/15 AFTER 100
2028/09/28 AFTER 47
2030/10/31 BEFORE 142
2027/01/05 BEFORE 146
2025/10/29 BEFORE 36
```

Sample Output

For each test case, your program must print the date the CDRL is due in the format YYYY/MM/DD. The month and day must contain two characters. If the CDRL date had to be adjusted due to being a weekend or holiday, print the adjusted due date with the word "ADJUSTED (+/-) 'X' DAY(S)" in all capital letters separated from the date by a space, where X is the number of days adjusted from the original calculated due date. If the date was only adjusted one day, state "DAY" rather than "DAYS".

```
2028/06/05 ADJUSTED + 1 DAY
2025/06/20 ADJUSTED - 1 DAY
2028/07/18
```

```
2025/10/30 ADJUSTED - 2 DAYS
2030/08/08 ADJUSTED - 1 DAY
2026/04/27 ADJUSTED + 2 DAYS
2028/11/14
2030/06/11
2026/08/12
2025/09/23
```

Problem 21: Voynich Manuscript

Points: 90

Author: Gary Hoffmann, Denver, Colorado, United States

Problem Background

The Voynich Manuscript is an enigmatic, hand-written manuscript dating from the 15th century. Named for its owner in the 20th century, Wilfrid Voynich, the document is often described as the most mysterious book in the world. It is filled with an as-of-yet undeciphered script known as "Voynichese" that does not appear to match any known language. This writing has stumped linguists, cryptographers, and codebreakers alike, and it remains unknown whether the text is a constructed language, some form of code or cipher, or if the entire document is simply a hoax perpetrated by Voynich himself.

Problem Description

Your team has been tasked with building a tool that can help to analyze blocks of text using an approach previously used to research the Voynich Manuscript.[1] This approach attempts to determine the informational value of each word that appears within the text by analyzing its frequency within specific sections of the text, compared against its frequency in sections of randomly shuffled versions of the text. For example, a math textbook might have the word "calculus" appear more frequently in some chapters than others; we can assume that's a key topic in the textbook.

Your team will be presented with a large block of text to analyze. The text will be divided into several sections, each containing an equal number of words. Before beginning your analysis, you should remove all punctuation and numbers from the original text (don't replace these characters with spaces; this may cause words you would read as separate words to be merged into a single word) and convert all letters to lowercase.

To calculate a word's entropy within a section of text, count the number of times the word appears in the relevant section (S) and how many times it appears in the entirety of the text, across all sections (N). Then input those values into this formula to calculate E_s, the entropy of the word for that section.

$$E_s = -\frac{S}{N}\log_2\frac{S}{N}$$

If a word does not appear in a particular section, its entropy for that section is zero (0). Once you've calculated a word's entropy within each section, add those values together to determine the word's overall entropy for the text ($E_{original}$).

[1] Montemurro MA, Zanette DH (2013) Keywords and Co-Occurrence Patterns in the Voynich Manuscript: An Information-Theoretic Analysis. PLoS ONE 8(6): e66344. https://doi.org/10.1371/journal.pone.0066344

Part of this analysis also requires calculating the average entropy for each word across an infinite number of randomly shuffled versions of the text ($E_{shuffled}$); that is, texts using the same words, but in a randomly determined order. Since you don't have an infinite amount of time, we've already calculated these values for you, and they will be provided in the input.

To calculate a word's informational value, count the number of times it appears across all sections of the text (again, N) and the total number of words in the text (C). Use those values along with the calculated ($E_{original}$) and provided ($E_{shuffled}$) entropy values to complete this formula:

$$I = \frac{N}{C}(E_{shuffled} - E_{original})$$

Finally, compare the informational values of each word and determine the top three keywords in each text; a higher value indicates a more important word.

Important note: If your programming language does not provide the native means to calculate a base-2 logarithm, you can calculate it using the natural logarithm (which it should support):

$$\log_2 X = \ln X / \ln 2$$

Sample Input

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

- A line containing two positive integers, separated by spaces:
  - S, the number of sections within the text to be evaluated
  - W, the number of words for which the value of $E_{shuffled}$ has been calculated.
- S lines, each containing a single section of the text to be analyzed. Lines may contain any printable characters, but will each contain the same number of words (within a test case). Lines will contain fewer than 2,000 characters.
- W lines, each containing a word that appears in the text (in lowercase letters), a space, and a decimal number representing that word's $E_{shuffled}$ value. Words that appear in the text that are not listed in this section have an $E_{shuffled}$ value of 0.0. Words will be listed in alphabetical order.

Due to its length, the sample input will not be replicated here. Please download the sample input from the contest website.

Sample Output

For each test case, your program must print the three words in the text with the highest informational values, on a single line and separated by spaces, in descending order of informational value. Print words using lowercase letters only.

```
chocolate pasta broccoli
```

Problem 22: Piece by Piece

Points: 95

Author: Brett Reynolds, Bethesda, Maryland, United States

Problem Background

Our ability to produces images of our environment has progressed incredibly since the invention of the daguerreotype in 1839. Current technology allows us to record images on virtually any scale - from the atomic structure of a substance to the vast expanses of the cosmos - and at incredible resolutions. At the time this problem was written, there were dozens of photos measuring in the hundreds of gigapixels; if printed at a standard resolution of 300 dpi, some of these images would cover more than a square kilometer.

Obviously, such large images are impractical for most uses; nobody's going to print out an image that covers an area larger than a city block, and most personal computers would struggle to even store such an image, let alone do anything with it. So, as with any software problem, the solution is simple: break the problem (image) into smaller parts that are easier to work with. "Image chunking" does exactly this; it divides a larger image into many smaller, uniformly sized "chunks" that accurately represent a tiny portion of the original image. This allows computers to more easily perform tasks on the image, as they don't need to load the entire image into memory; only a small bit of it at a time. However, it's important to ensure that the computer understands each chunk's location relative to the larger image, or your work becomes meaningless gibberish.

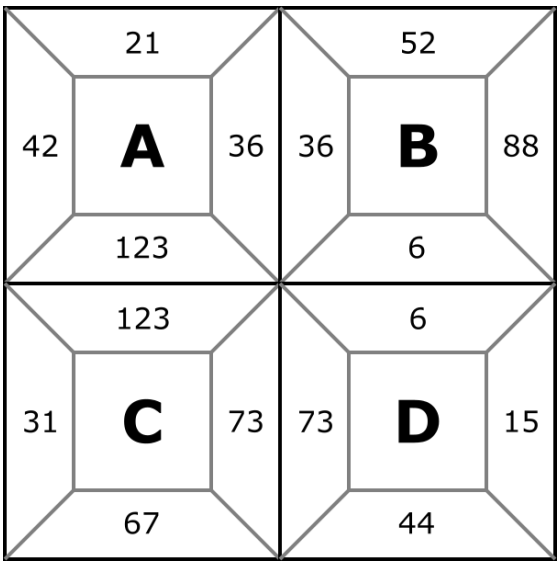
Problem Description

Lockheed Martin's Rotary and Mission Systems is working with the National Reconnaissance Office to develop a new radar imaging system that can cover a huge area. The images generated by the radar system are so large that they can't be stored as single files; they're saved as a number of image chunks along with the information needed to reconstruct the full image. Normally, this reconstruction is fairly straightforward, as each file is given a sequential name.

Unfortunately, someone on another team was careless about clicking links in an email, and exposed your work site to a virus that scrambled all of your files around. After containing the virus and assessing the damage, you've been able to determine that all of your data is still present, but some of it has been altered. Each of the chunk files was renamed, so you no longer have an easy method to reconstruct the full image. Additionally, it appears as though some of the images were rotated, so simply putting them in the correct order again won't be enough to get a usable image. You'll have to examine each of the files to determine how they fit together so that the NRO can perform a meaningful analysis on the data.

Each chunk file contains metadata that can be used to reconstruct the full image. Fortunately, it looks as though the virus managed to use your team's rotation utility program when corrupting the files, so

the metadata is still largely valid. Each side of each chunk is assigned a random number as an identifier. When two chunks are adjacent to each other, the touching edges will have the same identifier number; no other edges will have that identifier. See the image below for an example:



When a chunk appears on an edge of the image (or a corner), one (or two) of its edges will contain a unique identifier number, which does not appear in any other chunk.

As some of the chunks have been rotated, edges may not line up correctly in their current orientations; using the image above as an example, edge 36, shared between chunks A and B, may have been presented as chunk B's top edge originally.

Sample Input

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

- A line containing the following values, separated by spaces:
  - A positive integer, H, representing the height of the image in chunks. This value will be greater than or equal to 2.
  - A positive integer, W, representing the width of the image in chunks. This value will be greater than or equal to 2.
  - A single visible ASCII character, identifying a chunk that is known to be in the correct orientation (that is, the identified chunk does not require rotation; all other chunks may or may not require rotation). For a list of possible characters, see the US ASCII Table in the reference materials; spaces will not be used.
- H * W lines, each containing data extracted from a chunk file's metadata. This includes the following values, separated by spaces:
  - A single visible ASCII character that uniquely identifies the chunk. For a list of possible characters, see the US ASCII Table in the reference materials; spaces will not be used.

- An integer representing the identifier for the chunk's top edge.
- An integer representing the identifier for the chunk's right edge.
- An integer representing the identifier for the chunk's bottom edge.
- An integer representing the identifier for the chunk's left edge.

Within each test case, the resolution of the image (H * W) will not exceed 90 chunks.

```
2
2 2 A
A 21 36 123 42
B 36 52 88 6
C 73 67 31 123
D 6 15 44 73
3 3 %
$ 5 32 58 46
& 70 33 62 46
@ 71 44 80 1
% 80 19 83 58
^ 88 19 95 84
( 54 67 88 12
! 9 1 32 4
# 7 95 44 2
* 83 67 89 70
```

Sample Output

For each test case, your program must print H lines, each containing W ASCII characters representing the correct ordering of chunks within the full image. Lines should be printed in top-to-bottom order, with each line listing chunks within that row in left-to-right order.

```
AB
CD
!@#
$%^
&*(
```

Problem 23: Build-A-Title

Points: 100

Author: Michael Warner, Denver, Colorado, United States

Problem Background

While on a long car ride with your friends, you decide to start playing the game Build-A-Title together. However, you need an impartial judge that can accurately determine the winning answer.

Problem Description

Given a list of titles, determine the longest string that can be constructed by joining titles together on the common characters whenever the ending characters of one title equal the beginning characters of another title. The titles may be provided in any order. For example, with this list of titles:

- LA Law
- Courtroom Capers
- Law and Order
- Order in the Court
- Rick Steves Europe

The longest string would be:

- LA Law and Order in the Courtroom Capers

Each title may only be used once within each chain.

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 contain:

- A line with a positive integer, **N**, representing the number of titles to be provided.
- **N** lines, each containing a title with upper- and lower-case letters and spaces.

Due to its length, the sample input is provided on the next page.

```
2
3
Homestuck
Home Alone
Alone in the Dark
2
Driving Miss Daisy
The Daisy Chain
```

Sample Output

For each test case, your program must print the longest string that can be constructed by joining the titles together. If two or more possible strings have the same length, print the one that comes first in default string ordering.

```
Home Alone in the Dark
Driving Miss Daisy
```

Problem 24: Word Search

Points: 110

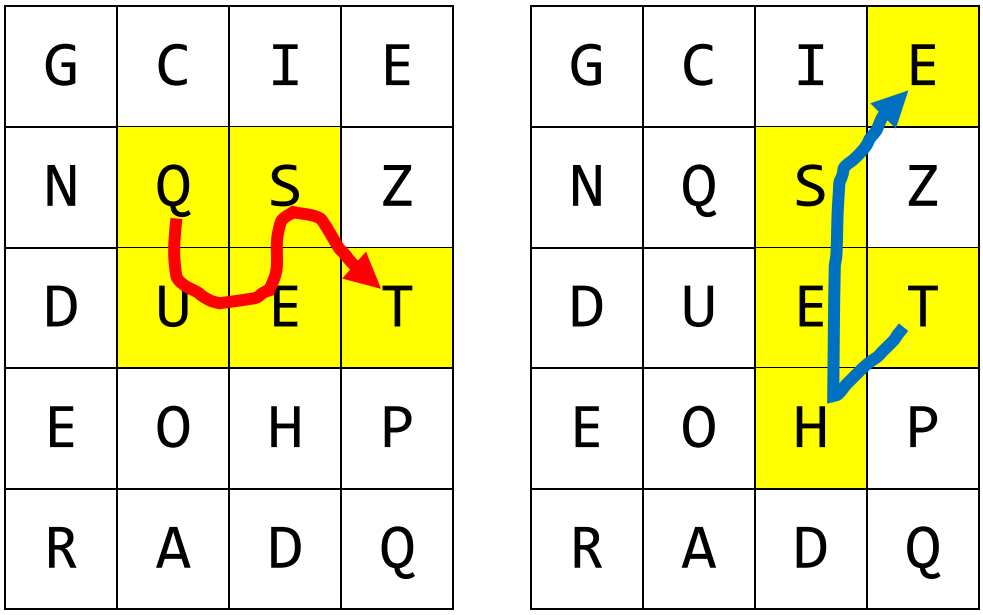
Author: Javier Jimenez, Marietta, Georgia, United States

Problem Background

Word search puzzles are a common form of puzzle, found in newspapers and restaurant kid's menus worldwide. Given a grid of seemingly random letters, you're expected to find a series of words hidden within the grid, written in any direction (forward, backward, up, down, or along any diagonal). Computers can solve such a puzzle easily, however; let's try making this a little more difficult.

Problem Description

In this version of a word search, words can change directions. As in a normal word search, each letter in a word will be adjacent to the one before it (in any of the eight surrounding positions), but that direction may not be consistent. For example, in the grid below, you can find the words "QUEST" and "THESE" by following the indicated paths (the same grid is shown on both sides):



If a word contains duplicate letters (for example, THESE contains two E's), each letter in the puzzle may only be used once within that word. As shown above, the final E must be pulled from the top row; it's not legal to double back and use the E in the third row twice. However, letters may be shared between words, as is done here with the shared E, S, and T between QUEST and THESE.

Design a program which can identify the path followed by each word given in such a puzzle by listing the coordinates of each letter within the word. The first (top-most) row is row 0, and the first (left-most) column is column 0, with numbers increasing as you move to the bottom or right of the grid,

respectively. Each test case is guaranteed to contain each target word only once; put another way, only one valid path exists for each word.

Sample Input

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

- A line containing three positive integers, separated by spaces, representing:
  - N, the number of words to locate within the puzzle grid
  - W, the width of the puzzle grid
  - H, the height of the puzzle grid
- H lines, each containing W uppercase letters, representing the puzzle grid
- N lines, each containing a word in uppercase letters at least five letters long.

```
1
5 4 5
GCIE
NQSZ
DUET
EOHP
RADQ
QUEST
ADORE
THESE
DUETS
THUNDER
```

Sample Output

For each test case, your program must print N lines indicating the paths taken through the puzzle grid by each target word. For each word provided (and in the order provided), your program should print the coordinates of each letter within the word, separated by spaces. Each coordinate should include the integer row number of the letter, a comma, and the integer column number of the letter. Row and column numbers range from 0 (the top- or left-most, respectively) to H-1 or W-1, respectively.

```
1,1 2,1 2,2 1,2 2,3
4,1 4,2 3,1 4,0 3,0
2,3 3,2 2,2 1,2 0,3
2,0 2,1 2,2 2,3 1,2
2,3 3,2 2,1 1,0 2,0 3,0 4,0
```

Problem 25: Did We Calculate Correctly?

Points: 120

Author: Chuck Nguyen, Rockville, Maryland, United States

Problem Background

Math is incredibly important; with math, we can do everything from managing our finances to traveling to the moon, Mars, and beyond! As a result, getting the correct answer to a math problem is often very important.

Problem Description

Given a mathematical problem, your program must determine if the problem was solved correctly. Problems may include any combination of the following operations in any order:

- Addition: `4+4=8`
- Subtraction: `4-4=0`
- Multiplication: `4*4=16`
- Division: `4/4=1` (you will never be required to divide by zero)
- Exponents: `4^4=256` (the value following the ^ will always be a positive integer)
- Square roots: `SQRT(4)=2` (the value subject to the square root will always be a positive integer contained within parentheses)

When an equation includes multiple operations, be sure to process them according to the usual order of operations. Equations may also include parenthesis to influence this ordering.

If using Python, we **strongly recommend** against using the `eval()` function to determine the correct result of the equation. Certain quirks regarding Python's order of operations may cause `eval()` to misinterpret some equations, resulting in a wrong answer. Our judging team will not provide any details of the test cases that are likely to encounter this problem.

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 mathematical equation. Equations will include at least one of the operations listed above in the formats indicated and may also include parenthesis. Any number may appear within equations except as otherwise indicated above. Each equation will end with an equals sign (=) and a numeric result, which may or may not be correct.

```
4
2*3+5*4=26
20/20*9=10
4*(5*-1)=-20
3^5-SQRT(20)=238
```

Sample Output

For each test case, your program must print the word "Correct" if the given result is correct, or the correct answer otherwise. Round results to three decimal places before comparing or printing them; do not print leading or trailing zeros, and do not print a decimal point when equations evaluate (or round) to an exact integer.

```
Correct
9
Correct
238.528
```