Code Quest Problem Packet

2022 Community College Outreach

Orange Technical College, Thursday, November 10, 2022

Valencia College, Friday, November 11, 2022

LOCKHEED MARTIN

Table of Contents

Welcome to Code Quest!

Lockheed Martin's Code Quest® is an annual competitive programming event held for high school students around the world. From its first contest in Fort Worth, Texas in 2012, the program has grown to include more than two dozen sites in six countries. These contests are developed and run by volunteer Lockheed Martin employees to showcase some of the innovative work done by our company and the many career paths available. Participants have the opportunity to start working towards those paths by applying for exclusive internships with real Lockheed Martin teams.

Lockheed Martin also offers a wide variety of opportunities for college students. You don't need a four-year degree to start a rewarding career at one of the country's largest defense contractors! In addition to traditional internships, paid apprenticeships offer on-the-job training to get you started in a new role. Mentoring programs within Lockheed Martin ensure that new employees get paired up with experienced co-workers to help them get started in their new position. If you're interested in continuing your education after starting at Lockheed Martin, the corporation is able to provide tuition reimbursement at a wide range of colleges and universities across the country, and can support working towards a large number of certification and continuing education programs as well.

In addition to Code Quest®, Lockheed Martin also offers other STEM outreach programs:

- CYBERQUEST is Lockheed Martin's annual cybersecurity competition, also for high school students. Challenges during this competition include cryptography and cryptanalysis, system penetration, incident response, and more. CYBERQUEST also offers internship programs within Lockheed Martin.
- Code Quest Academy offers in-classroom mentoring programs to help students improve their computer programming skills and get advice for participating in the Code Quest® competition and seeking a career in software engineering. Code Quest Academy also operates a website on which you can try all past Code Quest® problems on your own time, at https://lmcodequestacademy.com. You'll see these problems there in a few days!

If you're interested in learning more about, or even assisting with, any of these programs, please contact us at code-quest.gr-eo@lmco.com, or apply for an internship, apprenticeship, or full-time position at Lockheed Martin! We'd love to have you join our team.

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 volunteer Lockheed Martin employees responsible for judging the contest, however most of the judging is done automatically by the contest website. The contest website will compile and run your code, then compare your program's output to the expected official output. If the outputs match exactly, your team will be given credit for answering the problem correctly.

How is each problem scored?

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

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

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

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

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

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

Common errors may include:

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

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

I get an error when submitting my solution.

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

Can I get solutions to the problems after the contest?

Yes! Please email us at code-quest.gr-aero@lmco.com for solutions and other questions about these problems.

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" 15 minutes before the end of the contest, so keep working as hard as you can!

Mathematical Information

Rounding

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

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

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

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

- 1.5 rounds up to 2
- -1.5 rounds down to -2

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

Trigonometry

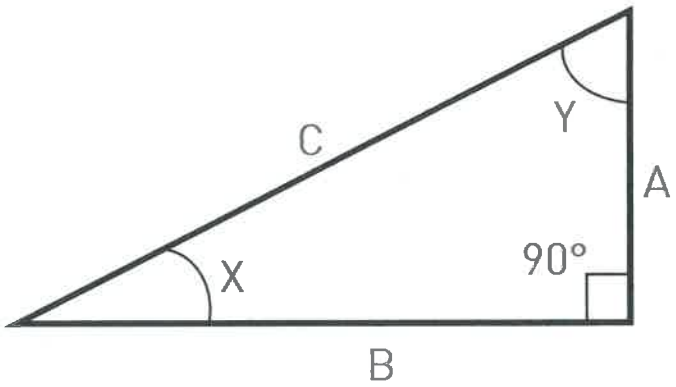
Some problems may require the use of trigonometric functions, which are summarized below. Most programming languages provide built-in functions for $\sin X$, $\cos X$, and $\tan X$; consult your language's documentation for full details. Unless otherwise stated in a problem description, it is *strongly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

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

$$X + Y = 90°$$

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

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



US ASCII Table

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

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

Terminology

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

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

DOMJudge Output

When judging your solutions, DOMJudge will return one of several results, as follows:

- CORRECT – Congrats! You've been awarded points and can work on other problems.
- WRONG-ANSWER – Your program did not produce the correct output for at least one of the test cases. When viewing the details, you may see that "Run 1" has a "CORRECT" result; this means that your program produced the wrong output for the official test cases, which are hidden from you. If "Run 1" shows a "WRONG-ANSWER" status, you should see your program's output as well as the ruling from the judging system. This may include messages such as:
    - "**`String token mismatch`**" – There was a difference in output, which should be indicated in the rest of the error message.
    - "**`Space change error`**" – This means there was an error in the whitespace, which may not be easily visible in the output. This message will be accompanied by numbers that correspond to the ASCII code points of the characters read. Common examples are:
        - "...**`got 32 expected 10`**" means the system read a space (ASCII 32) when it expected to find a line feed character (\n, ASCII 10).
        - "...**`got 13 expected 10`**" means the system read a carriage return (\r, ASCII 13) when it expected a line feed. Windows uses CRLF line endings (\r\n), but DOMJudge runs on Linux, and so uses LF line endings (\n only).
        - "...**`judge out of space, got 10 from team`**" means your output contains an extra blank line which wasn't expected.
- COMPILER-ERROR – Your program failed to compile and could not be run; the error reported should be visible within the submission details.
- TIMELIMIT – Your program was forcibly terminated after running for two minutes without stopping. All programs must finish execution within two minutes.
- RUN-ERROR – You program threw an unhandled error while running, preventing it from finishing. If the error occurred during the sample test cases, the error text may be visible in the submission details. Out-of-memory errors often result in this status, rather than "MEMORY-LIMIT" below.
- NO-OUTPUT – Your program did not print any input to the standard output channel. Make sure your program prints all output to the console, not to a file.
- OUTPUT-LIMIT – Your program produced more output than the system was able to handle. All output files are less than 10 MB in size, and most are less than 1 MB.
- MEMORY-LIMIT – Your program attempted to use more than the limit of 2 GB of memory (RAM) and was forcibly terminated as a result.

If you have any questions about the results you're getting, please send the judging team a clarification request and we'll be happy to help. Please note, however, that we cannot provide any details about the official test cases or your program's output for them.

Problem 1: What Is the Operation?

Points: 5

Author: Chuck Nguyen, Rockville, Maryland, United States

Problem Background

Have you ever wondered how a number came about? Here's your chance to solve the mystery.

Problem Description

Given a set of three numbers, your team must write a program that can determine the arithmetic operation required for the first two numbers to produce the third number. For example, given the numbers 7, 3, and 4 (in that order) the correct operation is subtraction; 7 - 3 = 4.

Possible operations may include:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/) (integer division, discarding any remainders)
- Modulo (%) (the remainder left after performing integer division)

In the event multiple operations could yield the third number (for example, 2 + 2 = 4 and 2 * 2 = 4), use the operation listed first on the list above.

Sample Input

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

```
2
7 3 4
2 2 4
```

Sample Output

For each test case, your program must print a single line containing one of the words, "Addition," "Subtraction," "Multiplication," "Division," or "Modulo," indicating the operation to use with the first two input numbers to produce the third input number.

```
Subtraction
Addition
```

Problem 2: Easter Sunday

Points: 5

Author: Jonathan Tran, Dallas, Texas, United States

Problem Background

Easter Sunday is a major religious holiday celebrated by Christian faithful around the world. While the holiday always occurs on a Sunday, the specific date can vary widely from year to year, and even between different countries and denominations of Christianity. The main reason for this is that the date of the holiday is based upon a lunar calendar, generally occurring on the next Sunday after a certain full moon. As a result, the holiday could take place anywhere from late March to late April.

As we said, the actual date on which Easter is observed is largely up to religious authorities, but there are ways to calculate when Easter is expected to take place.

Problem Description

In 1800, mathematician Carl Friedrich Gauss developed an algorithm for exactly this purpose. His algorithm is outlined below:

$$
\begin{aligned}
y &= the\ year \\
a &= y\ mod\ 19 \\
b &= y\ mod\ 4 \\
c &= y\ mod\ 7 \\
k &= floor\left(\frac{y}{100}\right) \\
p &= floor\left(\frac{13+8k}{25}\right) \\
q &= floor\left(\frac{k}{4}\right) \\
m &= (15-p+k-q)\ mod\ 30 \\
n &= (4+k-q)\ mod\ 7 \\
d &= (19a+m)\ mod\ 30 \\
e &= (2b+4c+6d+n)\ mod\ 7 \\
f &= (11m+11)\ mod\ 30
\end{aligned}
$$

After finally calculating d, e, and f, you can use these rules to calculate the actual date:

- Calculate $22+d+e$ to get the date. If less than or equal to 31, the date is in March. If greater than 31, subtract 31; that is now the date in April.
- If the calculated date is April 25th, $d = 28$, $e = 6$, and $f < 19$, change the date to April 18th instead.
- If the calculated date is April 26th, $d = 29$, and $e = 6$, change the date to April 19th instead.

Given a calendar year, use Gauss's algorithm to calculate the expected date of Easter according to the lunar calendar.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include a single line containing an integer representing a calendar year greater than 1900.

```
3
2021
2022
2019
```

Sample Output

For each test case, your program must print a single line containing the expected date of Easter Sunday according to Gauss's algorithm, in YYYY/MM/DD format.

```
2021/04/04
2022/04/17
2019/04/21
```

Problem 3: Robot Petsitter

Points: 10

Author: Anthony Vardaro, Dallas, Texas, United States

Problem Background

Suppose you wanted to automate your chores around the house, and you decided to build a robot that would walk your dog around your neighborhood. It's critical that the robot returns to your house, otherwise your dog might get lost. Prior to its journey, the robot downloads a sequence of moves that describes its path during the adventure. You must design an algorithm that analyzes the robots move sequences and determines whether the robot's path will guide it back home properly.

Problem Description

The robot moves as though on a grid, where your house, and the robot's starting point, are at the origin point (0,0). It is only capable of moving one block at a time along the grid's axes; up, down, left, and right. Given a set of directions, you need to confirm that after following those directions, the robot has returned to your house.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include a single line of text containing at least one uppercase letter; any of D, L, R, and/or U (representing the directions down, left, right, and up, respectively).

```
4
LLRR
ULDR
RRDRULDDL
U
```

Sample Output

For each test case, your program must print a single line containing the word "TRUE" if the given directions return the robot to its starting point, or "FALSE" if they do not.

```
TRUE
TRUE
FALSE
FALSE
```

Problem 4: Are You a Spy?

Points: 15

Author: Steve Gerali, Denver, Colorado, United States

Problem Background

You have been hired by the Central Intelligence Agency to help CIA agents identify each other in the field using a secret strategy. Instead of pre-defined code phrases - that's Hollywood nonsense! - the first agent will say a greeting to the second agent. The second agent must respond using words that contain only letters that appear in the original greeting.

Problem Description

You will be given a series of greetings and responses. The greetings and responses can be of any length (not necessarily the same length), and may contain numbers, letters, spaces, and/or punctuation marks.

If all of the letters in the response also appear in the greeting, your program will need to display "That's my secret contact!" You can ignore any numbers, spaces, and punctuation marks when making your comparisons, and can ignore the case of letters.

If the response contains any letter that is not present in the greeting, you should print "You're not a secret agent!" instead.

For example, if the first agent says "Hello there" and receives a reply of "General Kenobi," the second person is not a secret agent (but definitely a Star Wars fan); the letters A, B, G, I, K, and N appear in the response, but do not appear in the greeting. On the other hand, if the agent says "It's raining" and is told "I sang in a train" in response, they can continue the conversation with confidence that they are speaking to another CIA agent.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include a single line with two sentences - the greeting and response, respectively - separated by a pipe character (|). Each sentence may be of any length and may contain any printable character except pipes.

```
4
Good evening.|Hello there.
Dogs can't fly without umbrellas.|Bread is a stale food.
It's nice to meet you, George.|Greetings to you too, Tim!
It's raining.|I sang in a train.
```

Sample Output

For each test case, your program must print a single line with the sentence "That's my secret contact!" if the response is valid, or the sentence "You're not a secret agent!" otherwise.

```
You're not a secret agent!
That's my secret contact!
That's my secret contact!
That's my secret contact!
```

Problem 5: Discovering Planets

Points: 15

Author: Sowmya Chandrasekaran, Sunnyvale, California, United States

Problem Background

A planet has been discovered by a group of astronomers! Now they must work to determine if the planet can support life. There are many determinants in arriving at a conclusion, including the presence of water, and reasonable temperature levels and climates. Astronomers can use decision trees to help inform their conclusions. Decision trees are flowchart-like structures that map out possible outcomes of a series of related actions. They are frequently used to arrive at a decision and allow the user to weigh possible actions and their consequences against one another based on their probabilities.

Problem Description

For a planet to be considered habitable, it must meet these conditions:

- **Temperature**: The advanced life that we know about depends upon water. As a result, one condition for the habitable zone is that water can exist in liquid form. This requires a temperature range between 0°C and 100°C.
- **Presence of Water**: Again, water is essential to life as we know it. Having a good temperature means little if there's no water on the planet. Astronomers can use advanced telescopes to identify the presence of water vapor in an atmosphere, or ice or water on the planet's surface.
- **Survivable Surface**: Earth's global magnetic field shields surface life from the lethal effects of charged particles in the solar wind and in cosmic rays. The presence of such a field is essential to a planet's ability to support life.
- **Circular Orbits**: A stretched-out, heavily elliptical orbit results a varying climate, which could prevent the evolution of complex organisms, and even the origin of life. A more stable and less elliptical orbit ensures more consistent conditions on the planet. This is represented by the orbit's eccentricity; an eccentricity of 0.0 indicates a perfectly circular orbit, whereas a value closer to 1.0 indicates a more elliptical orbit. An ideal orbit for a habitable planet would have an eccentricity less than 0.6.

Your team is working with the National Aeronautics and Space Agency (NASA) and the European Space Agency (ESA) to evaluate exoplanets for habitability. You'll be provided with information about each planet on each of the points above, and must construct a decision tree to determine if the planet is habitable or not; and if not, the primary reason why.

Sample Input

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

- A decimal value representing the average estimated temperature on the planet's surface, in degrees Celsius (°C).
- A Boolean value indicating the presence (true) or absence (false) of water in some form.
- A Boolean value indicating the presence (true) or absence (false) of a planetary magnetic field.
- A decimal value, between 0.0 and 1.0 exclusive, representing the eccentricity of the planet's orbit.

```
4
13.81 true true 0.0167
-12.51 true true 0.6124
53.4 false true 0.5116
23.1 true true 0.7234
```

Sample Output

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

- If the planet has a temperature above 100°C, print "The planet is too hot."
- Otherwise, if the planet has a temperature below 0°C, print "The planet is too cold."
- Otherwise, if the planet does not contain water, print "The planet has no water."
- Otherwise, if the planet does not have a magnetic field, print "The planet has no magnetic field."
- Otherwise, if the planet's orbit has an eccentricity greater than 0.6, print "The planet's orbit is not ideal."
- Otherwise, print "The planet is habitable."

```
The planet is habitable.
The planet is too cold.
The planet has no water.
The planet's orbit is not ideal.
```

Problem 6: By the Book

Points: 15

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

Problem Background

When searching for books in the library, it can sometimes be difficult to be certain if you've gotten the correct book. Particularly with reference books or other textbooks, titles can be the same or similar and it's not always easy to tell who the author is. Fortunately, publishers have developed a way to be certain you've got not only the correct book, but the correct version of it: the International Standard Book Number, or ISBN.

An ISBN is a 10- or 13- digit code (books published since 2007 have 13-digit codes) that is supposed to uniquely identify a particular edition of a book (although in practice, there are some overlaps). Since accurately remembering or writing down a long string of numbers is not guaranteed, the ISBN contains a built-in error-checking mechanism. The last digit in an ISBN is known as a "check digit" and can be used to verify that all of the preceding digits represent a valid ISBN.

Problem Description

Some Lockheed Martin sites maintain a library of technical books for employees to check out; this helps employees learn about technologies they may not be familiar with. Unfortunately, your site suffered a fire recently, and you've volunteered to help assess the damage and try to recover the collection. The library has a collection of books (all published before 2007) with labels that were damaged in the fire. Someone has already attempted to restore the ISBN numbers on these labels, but it quickly became apparent that not all of the numbers are correct. You have been asked to write a program that can read in ISBN numbers from these labels and determine if the numbers are valid or invalid. Books with invalid ISBNs will be set aside to be manually researched and corrected.

To calculate the check digit of an ISBN-10, each of the first nine digits is multiplied by an "integer weight," a number calculated by subtracting the index of the digit from 10. These products are then added together to form a single number. The check digit is the number that must be added to that sum to reach a multiple of 11 (if that number is 10, the check digit is the letter 'X').

For example, consider the ISBN-10 number 0-306-40615-2:

$$S = \sum_{i=0}^{8} x_i \times (10 - i); \;\; x = \{0,3,0,6,4,0,6,1,5,2\}$$

$$S = \big(0 \times (10-0)\big) + \big(3 \times (10-1)\big) + \big(0 \times (10-2)\big) + \big(6 \times (10-3)\big) + \big(4 \times (10-4)\big) + \big(0 \times (10-5)\big) + \big(6 \times (10-6)\big) + \big(1 \times (10-7)\big) + \big(5 \times (10-8)\big)$$

$$S = (0 \times 10) + (3 \times 9) + (0 \times 8) + (6 \times 7) + (4 \times 6) + (0 \times 5) + (6 \times 4) + (1 \times 3) + (5 \times 2)$$

$$S = 0 + 27 + 0 + 42 + 24 + 0 + 24 + 3 + 10$$
$$S = 130$$

The weighted sum of this ISBN is therefore 130. 130 is not divisible by 11; the next highest multiple of 11 is 132 (11 * 12 = 132). Since 132 - 130 is 2, the check digit for this ISBN should be 2. This matches the last digit in the ISBN we were investigating, so this ISBN number appears to be valid.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include a single line containing an ISBN-10 number to be checked. ISBNs can include numeric digits and the uppercase letter X.

```
3
0306406152
0306401652
080442957X
```

Sample Output

For each test case, your program must print, on a single line, the word VALID if the given ISBN number has a valid check digit, or the word INVALID if it does not.

```
VALID
INVALID
VALID
```

Problem 7: Natural Boost

Points: 20

Author: Matthew Ward, Huntsville, Alabama, United States

Problem Background

We all know that Atlas rockets use chemical boosters to burn fuel and generate thrust, but did you know that the Earth also plays a big part in getting to orbit? When a rocket lifts off, it doesn't go straight up; it tilts its nose toward the Earth slightly so that it starts flying sideways. Getting to Orbit is about going so fast parallel to the surface of the Earth that the Earth's surface curves away from you faster than you can fall towards it.

As a result, the closer you are to the equator, the greater the speed boost you get from the rotation of the Earth. The United States takes advantage of this by launching most of its rockets from Cape Canaveral in Florida, Vandenberg Air Force Base in Southern California, and Wollops Island in Virginia.

Problem Description

Lockheed Martin Space is designing a new simulation tool to estimate how much fuel a rocket will need to reach orbit. Your team's job is to design an algorithm that can calculate the rocket's initial lateral speed based on the latitude of the launch site.

For this problem, you can assume that the Earth is a perfect sphere with a radius of 6,370,000 meters which rotates exactly around its north-south axis once every 24 hours (or 86,400 seconds). None of this is actually true in practice - the Earth is squashed and tilted, and days aren't exactly 24 hours long - but we're just getting estimates, so we don't really need exact numbers.

Given the latitude of the launch site in degrees (0° is the equator; 90°is the North Pole), calculate how much velocity will be applied to the rocket simply from the rotation of the Earth.

These equations may help along the way, along with the trigonometric functions provided in the reference materials:

- Radius of a circle, in meters: r
- Time, in seconds: t
- Distance, in meters: x
- Circumference of a circle, in meters: $C = 2\pi r$
- Velocity, in meters per second: $v = {}^{x}/_{t}$

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 decimal value between -90.0 and 90.0 inclusive representing the degrees of latitude of the launch site.

```
5
28.3922
34.7420
-37.9402
5.0500
0.0000
```

Sample Output

For each test case, your program must print a single line containing an integer value, representing the floor of the rocket's velocity due to the rotation of the Earth, in meters per second.

```
407
380
365
461
463
```

Problem 8: Augmenting Reality

Points: 25

Author: Kelly Reust, Denver, Colorado, United States

Problem Background

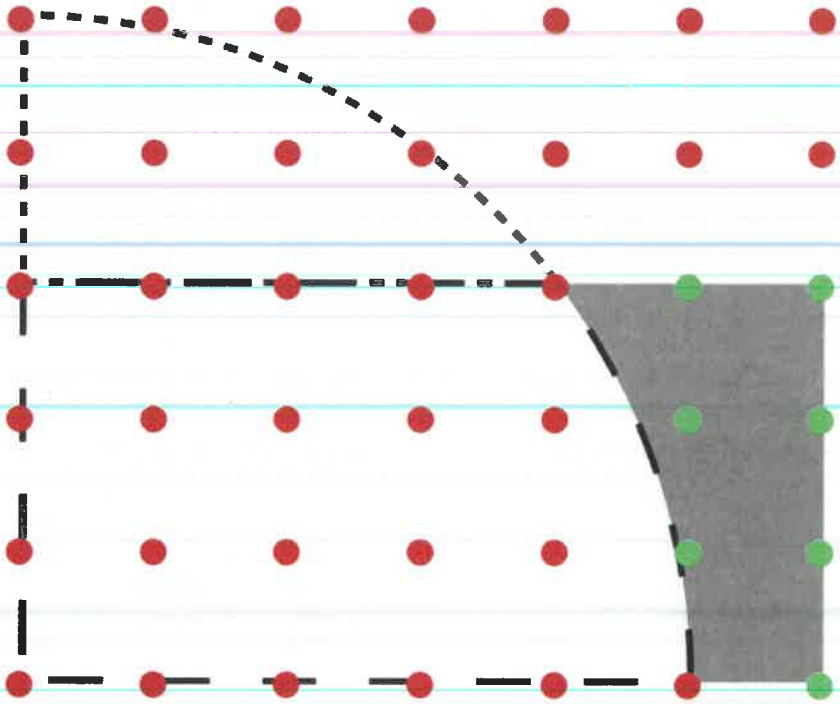
Lockheed Martin is working on a new augmented reality system to help soldiers identify parts to be installed on military aircraft undergoing maintenance. Maintainers will wear a headset that includes a camera and two small screens in the form of eyeglasses. When a maintainer picks up a part, the system will use the images captured by the camera to attempt to identify the part. If the part is successfully identified, a small overlay will appear on the screens providing information about the part in question. From the maintainer's perspective, the overlay will appear to be "hovering" over the part they're holding, allowing them to quickly identify the correct parts to install without having to look up serial numbers on a conventional computer system.

However, your team's goal is to be helpful to the maintainers, and not hinder their work. The overlays you create need to be large enough to include any information they might need, but shouldn't be so large that they cover the user's entire field of vision. Your team has decided to define an area of the screen that should always remain clear to avoid blinding the user; any overlay information must be displayed outside of this area.

Problem Description

The "no-overlay" zone for your team's AR system will be defined as a circle, centered on the lower left corner of the screen. The overlays themselves will be rectangles, positioned so their lower left corner is in the lower left corner of the screen. Any portion of the overlay that falls within the circular "no-overlay" zone should not be rendered. Your team needs to write a program that determines which pixels should be rendered to avoid infringing on the "no-overlay" zone.

For example, see the example screen layout displayed on the next page. This shows a situation where the "no-overlay" zone has a radius of 5 pixels, and the overlay is 6 pixels wide by 3 pixels high.



The curved dotted line represents the border of the "no-overlay" zone. Since it's defined by a circle centered on the screen's lower left corner, we only see the upper right quadrant of the circle. The overlay is represented by the shaded area; even though it's defined by a rectangle anchored to the screen's lower left corner, we only see the portions that extend beyond the "no-overlay" zone. The colored dots represent the pixels, positioned at the integer X,Y coordinates on a grid where the origin is the lower-left corner. The green dots are the only pixels within the rectangular overlay and strictly outside of the circular "no-overlay" zone. The pixels at 4,3 and 5,0 appear on the edge of the circle, and so are not rendered.

Given the radius of the "no-overlay" zone and the dimensions of the overlay's rectangle, your team needs to develop an algorithm to determine which pixels on the screen need to be rendered. Pixels are represented by non-negative integer X,Y coordinates, where the origin (0,0) is the lower left corner of the screen. Higher X values appear further to the right on the screen, and higher Y values appear further up on the screen.

Sample Input

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

- R, a positive number representing the radius of the circular "no-overlay" zone, centered on the lower-left corner of the screen
- W, a positive number representing the width of the overlay rectangle, as measured from the lower-left corner of the screen

- H, a positive number representing the height of the overlay rectangle, as measured from the lower-left corner of the screen

```
2
5 6 3
4 5 3
```

Sample Output

For each test case, your program must print a list of non-negative integer X,Y coordinates that fall within the defined rectangle but do not fall within the defined circle. Each coordinate should be printed on a separate line, and should be sorted in increasing order; first by the X coordinate, then by the Y coordinate. Numbers should be separated by a comma.

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

Problem 9: Halves & Doubles

Points: 30

Author: Dan Knudson, Milwaukee, Wisconsin, United States

Problem Background

Multiplying small numbers isn't too difficult, but multiplying very large numbers can tie your brain in knots as you try to keep track of all the digits in your head. However, there's an ancient method of multiplication that might be somewhat easier to use. Regardless of what numbers you're trying to multiply, it only requires multiplying by 2, dividing by 2, and addition!

Problem Description

The "Halves and Doubles" method of multiplication follows a set of rules:

1. Reduce the first number by half. If there's a remainder (e.g. 5 / 2 = 2 ½), throw away the half.
2. Double the second number.
3. Write down the pair of numbers, keeping them in the same order.
4. Repeat steps 1 through 3 until the first number equals 1.
5. Check the list of numbers you've written. Cross out any pairs where the first number is even.
6. Add together all of the second numbers to get your answer.

For example, if multiplying the numbers 10 and 20 using this method, we start by building a list of pairs of numbers (steps 1 through 4):

10	**20**
5	**40**
2	**80**
1	**160**

Now we cross out any pairs where the first number is even:

~~**10**~~	~~**20**~~
5	**40**
~~**2**~~	~~**80**~~
1	**160**

The remaining numbers in the right column get added together for our answer: 40 + 160 = 200 = 10 * 20.

For this problem, your team needs to write a program demonstrating the Halves & Doubles method of multiplication.

Sample Input

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

```
3
10 20
11 6
60 200
```

Sample Output

For each test case, your program must print one line for each pair of numbers created while performing this process, starting with the initial pair provided in the input. Numbers should be separated by spaces and be retained in the correct order. If the first number in a pair had a discarded remainder following its division, print an asterisk (*) after that number (before the space). If the first number in a pair is even, print a space and three asterisks (*) after the second number to indicate the row is to be crossed out. Finally, print the result of the multiplication on a separate line after the list of pairs.

```
10 20 ***
5 40
2* 80 ***
1 160
200
11 6
5* 12
2* 24 ***
1 48
66
60 200 ***
30 400 ***
15 800
7* 1600
3* 3200
1* 6400
12000
```

Problem 10: Keeping Inventory

Points: 40

Author: Steve Brailsford, Marietta, Georgia, United States

Problem Background

Keeping military equipment in working order - be it an airplane, tank, ship, or something else - is a critical task to ensure that the military is able to respond to any situation as quickly as possible. However, all of those airplanes, tanks, ships, and so on are each comprised of thousands of parts; keeping track of them all can be a HUGE task. This is further complicated by the fact that most of those parts are identified by largely incomprehensible part numbers.

Fortunately for contractors like Lockheed Martin, the United States Military has a solution for that. A series of standing orders from the Department of Defense, known as military standards, dictate how parts like these should be managed, all the way down to how to sort a list of part numbers.

Problem Description

Your team is working with Lockheed Martin's Rotary and Mission Systems division to create a new maintenance inventory system for a branch of the US Military. One of your requirements is to provide a list of all parts currently held in inventory. After discussing the requirement with the customer, you learn that the military standard MIL-DTL-38807D provides detailed instructions on how to sort a list of part numbers. Sorting the numbers correctly is important, as if parts are listed out of order, a maintainer may think they're out of stock of a part when they're not.

The full standard is very long and boring, and so we won't quote it here, but the important part (as far as your team is concerned) says that part numbers should be sorted by the characters they contain, from left to right, as follows:

1. Diagonals (represented as a forward slash, /)
2. Periods (.)
3. Dashes (-)
4. Uppercase letters A to N and P to Z, in alphabetic order
5. Numerals from 0 to 9 in increasing order. The uppercase letter O is treated the same as a numeral zero (0).

For example, the part number A-113 should appear before the part number ABC; even though they have the same first character, A-113 has a dash as the second character, which ranks above any letter.

Part numbers will only contain the characters listed above, and will not start with diagonals, periods, nor dashes.

Sample Input

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

- A line containing a positive integer, X, indicating the total number of types of parts in inventory
- X lines, each containing the part number of a part in inventory, containing uppercase letters, numbers, periods (.), dashes (-), and/or slashes (/). Within a test case, all part numbers will be unique.

```
1
5
ABC
8CX.12/3
A-113
A.234
GQ/BBQ-12
```

Sample Output

For each test case, your program must print the given list of part numbers, one part number per line, in the order dictated by the military standard.

```
A.234
A-113
ABC
GQ/BBQ-12
8CX.12/3
```

Problem 11: Bankrupt on Fuel

Points: 50

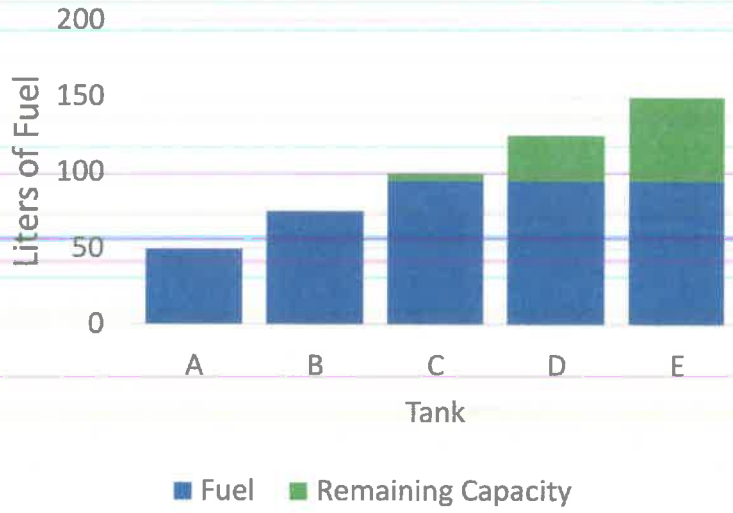
Author: Wojciech Koziot, Mielec, Poland

Problem Background

A bankruptcy or claims problem is a type of problem in which you need to fairly distribute some sort of finite, divisible resource amongst a number of recipients that each have a differing claim on that resource. The name comes from the similarities to financial bankruptcy; if someone doesn't have enough money to pay all of their debts, they are bankrupt. They must work with the legal system to determine a way to pay each of their creditors what they are owed. This sort of problem also occurs in computer operating systems. Each program running on a computer requires a certain amount of memory to be able to operate, but the computer only has a limited amount. The operating system is responsible for determining which programs get access to memory, and how much.

There are many ways to determine which of the claimants gets access to resources, and in what amounts; one of these methods, the "constrained equal awards rule," closely mirrors how a liquid fills a series of open containers.

Problem Description



Lockheed Martin Space is working on building a new rocket launch facility for launching government satellites. This facility will contain a number of launch pads designed for different types of rockets. Each launch pad will have its own tank to hold rocket fuel; since the rockets launched at different pads will be different sizes, these tanks have different sizes as well. All of the tanks are connected to a central refueling system, which equally distributes fuel between all of the launch pads. Once a tank is filled, it stops receiving fuel, but all other unfilled tanks will continue to receive fuel. When the fuel supply is cut off, any unfilled tanks should have the same amount of fuel. The graph above shows an example, where five tanks with capacities of 50, 75, 100, 125, and 150 liters were filled with 410 liters of fuel; the smaller tanks are full, and the larger ones each hold 95 liters.

Unfortunately, these tanks are very large, and with recent supply chain issues, the facility supervisor is uncertain that it will be possible to obtain enough fuel to completely fill all of the tanks. She has asked your team to design a simulation tool that can indicate the expected amount of fuel in each tank, given

their capacities and an amount of fuel to be distributed amongst them. To ensure that your tool is accurately reporting those amounts, she's also asked that you return the results in a fractional form, rather than rounding off decimal places.

Sample Input

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

- A line containing two positive integers, separated by spaces:
  - X, representing the amount of fuel available, in liters
  - N, representing the number of tanks to be filled
- A line containing N positive integers, separated by spaces. Each integer represents the capacity, in liters, of a fuel tank. The sum of all integers in this line will be greater than X.

```
3
203 3
101 50 300
199 2
100 102
410 5
50 75 100 125 150
```

Sample Output

For each test case, your program must print the amount of fuel each tank will contain when fueling is complete. List tanks in the order in which they were presented in the input and separate each value with a space. Print integers where possible; where not possible, use a simplified fractional format, including a numerator, a forward slash (/), and a denominator, such that the numerator and denominator are co-prime (their greatest common divisor is 1).

```
153/2 50 153/2
199/2 199/2
50 75 95 95 95
```

Problem 12: Old-Timer Texting

Points: 60

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

Problem Background

You're visiting your grandparents, but are starting to get bored. You pull out your phone to text your friends to see what they're up to, but your grandfather sees what you're doing. "You kids these days!" he complains. "You don't know how good you have it with your touch screens and smart phones. Back in my day, when we wanted to text someone, we had to use a number pad to type! We had to hit each button several times until it showed the letter we wanted. Texting took forever!" Fortunately, your grandmother shushes him before he starts complaining about other ancient technologies.

Once you get home, however, you start to wonder... just how difficult *was* texting back in the olden days?

Problem Description

As noted above, when SMS messaging - or texting - was first introduced, full keyboards were extremely uncommon on mobile devices. Most phones simply had a keypad for dialing phone numbers, and a few other buttons for navigating menus and actually answering the phone. To support texting, each number button was associated with a set of letters.

Pressing a number button once would type the first letter in that button's set; quickly pressing it twice would type the second number, and so on. The typed letter would be locked into place after a short delay, or if another button was pressed. For example, given the standard US keypad layout shown at right, typing "hello world" would require pressing:

1	2 ABC	3 DEF
4 GHI	5 JKL	6 MNO
7 PQRS	8 TUV	9 WXYZ
*	0 [space]	#

44-33-555-555-666-0-9-666-777-555-3

You mention this to your friends, and they find the whole process both inefficient and hilarious. You decide to write a program to translate text into keypad texting commands just to see how bad it really was. If nothing else, it might get your grandfather to stop complaining so much.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include a single line of text comprised of lowercase letters and spaces, up to 160 characters long.

```
3
hello world
lockheed martin
code quest
```

Sample Output

For each test case, your program must print a single line indicating the buttons that must be pressed on the keypad displayed above to produce the given text string. For each character in the original message, show the number that must be pressed, repeated to indicate the number of times that button must be pressed to produce that character. Separate the representation for each character with a dash (-).

```
44-33-555-555-666-0-9-666-777-555-3
555-666-222-55-44-33-33-3-0-6-2-777-8-444-66
222-666-3-33-0-77-88-33-7777-8
```