



Tetrahedral Stacks of Cannonballs



WALL•E™©, as he cleans up and organizes the depopulated Earth, has come upon some Civil War memorials. He is consolidating the cannonballs into one location, and decides to use pyramids with triangular bases rather than ones with square bases.

In Civil War memorials with cannons and stacks of cannonballs, the cannonballs were sometimes stacked as a four-sided pyramid, with the base as a square of cannonballs with n balls on each side. An alternative is to stack them in a three-sided pyramid, which is in fact one of the Platonic solids, a tetrahedron.

This tetrahedron of cannonballs has a base that is an equilateral triangle of cannonballs with n balls on each side. The number of balls in that triangle is given simply by adding together the numbers from 1 to n . On top of each layer (starting from the base) is a triangle with one less ball on each side, up to the top-most layer with a single ball.

Given the number of cannonballs on each side of the base, compute the total number of cannonballs in the entire tetrahedral stack.

Input

The first line contains a single number n , giving the number of tetrahedral problems posed, for a maximum of 100 problems. Following that are exactly n lines, each with a single number giving the number of cannonballs on each side of the base for a tetrahedron of cannonballs, a number greater than 0 and less than 1000.

Output

For each problem, output the problem number (starting from 1), a colon and a blank, the number of cannonballs on each side of the base, one blank, and finally the total number of cannonballs in the tetrahedron.

Sample input

```
6
1
2
3
5
27
999
```

Sample output

```
1: 1 1
2: 2 4
3: 3 10
4: 5 35
5: 27 3654
6: 999 166666500
```

“Robot Roll Call – Cambot...Servo...Gypsy...Croooow”

Mystery Science Theater 3000 is about to start, which means it's time for “Robot Roll Call”, where the name of each robot is called out, as per the list received from Earth. The expectation is that if a robot is there, it will respond by adding its name to a data stream which is then sent back to Earth. Unfortunately today, once the roll is received, communication with Earth is temporarily lost. In the meantime, the robots that are present for roll call have saved their names to the data stream. However, lots of other things are also being saved to this same stream. To help extract data later, any data placed in the stream is separated by whitespace. Once the communication problems are resolved, the contents of this stream are relayed to Earth.



Your task is as follows. Given a list of names for roll call, you must scan the accompanying data stream and determine if a given name is there. For each name that is in the roll call, report whether or not that name was in the data stream. For a name to be a match, it must appear **EXACTLY** as shown in the roll. This means a match is case-sensitive and sub-string matches are not allowed.

Input

The first entry in the file will be an integer value t ($t > 0$) that represents the number of test data sets the file contains. Following this entry, will be t test sets. Each test set will start with an integer n ($0 < n < 26$) representing the number of names in the roll. On the lines that follow will be n entries, one per line, containing the individual names in the roll. No name will have more than 25 characters. Names will only contain the characters A-Z, a-z, and 0-9. Names will be unique.

Following the names will be an integer d ($0 < d < 100$) representing the number of lines in the data stream. On each subsequent line will be the characters that make up the data stream. Each line of the data stream will contain at least one character and at most 100. Furthermore, the data on a given line will be separated by whitespace (space, tab, or combination of the two). Finally, any names from the roll that might occur as part of the data stream will be found on one line (a name will not be split across lines).

Output

Write the test set number (starting at 1) on a line of its own, followed by the names in the roll and whether or not a given name was found in the data stream. Each of these names should occur on a line of their own. Add a blank line to the end of each test case. Output format must be identical to the sample output shown below.

Sample Input

```
2
4
Gypsy
TomServo
CrowTRobot
Cambot
2
Manos Torgo Joel 101010 Gypsy tomservo
Fugitive Alien Time of the Apes crowTrobot Cambot
2
R2D2
C3PO
1
Boba Fett c3Po R2D2 Jar Jar Binks Luke give in to the dark side
```

Sample Output

```
Test set 1:
Gypsy is present
TomServo is absent
CrowTRobot is absent
Cambot is present
```

```
Test set 2:
R2D2 is absent
C3PO is absent
```

Cave Crisis

R2D2 was exploring a tunnel when a cave-in suddenly occurred. Oh no, is he trapped?

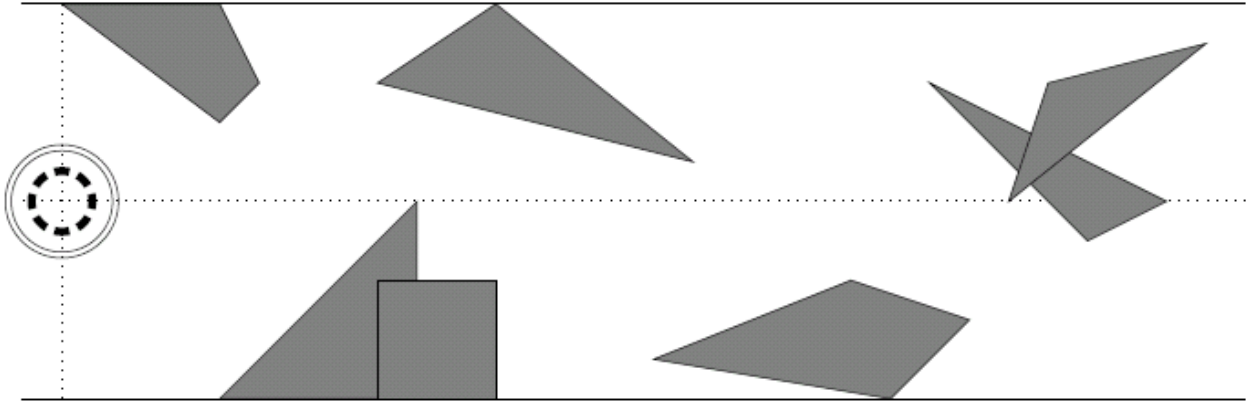


Figure1: Overhead view of the cave crisis from the third example test case.

From an overhead view, we can see all the obstacles (debris) on a two-dimensional Cartesian plane. The tunnel is w cm wide, bounded by the lines $y = w/2$ and $y = -w/2$. R2D2 starts at $(0, 0)$, and has a perfectly circular footprint of radius r . The exit of the tunnel lies to the right of the line $x = 1000$. Between R2D2 and the exit lie a number of polygonal obstacles.

Is it possible for R2D2 to navigate between the obstacles and make it to the exit?

Input

The input file will contain multiple test cases. Each test case begins with a single line containing an even integer w ($2 \leq w \leq 1000$), the width of the tunnel, and an integer N ($0 \leq N \leq 100$), the number of obstacles. The next N lines each contain the description of a single obstacle. The i _{th} obstacle is a simple polygon, specified on a single line containing an integer n_i ($3 \leq n_i \leq 10$), the number of vertices, followed by n_i pairs of integers, x_{ij} and y_{ij} ($0 \leq x_{ij} \leq 1000$ and $-w/2 \leq y_{ij} \leq w/2$ for $j = 1, \dots, n_i$), specifying the coordinates of the vertices in counterclockwise order. Note that obstacles in the input may touch or even overlap. However, you are guaranteed that R2D2's initial location will not touch or overlap any obstacle. The vertices of each polygon are unique, no two nonconsecutive edges of the polygon intersect (even at their endpoints), and each polygon is guaranteed to have nonzero area. The end-of-input is denoted by an invalid test case with $w = N = 0$ and should not be processed.

Output

For each input test case, you must determine the maximum radius $r > 0$ that R2D2 could have and still be able to plan a path from his starting location $(0, 0)$ to the tunnel exit without overlapping with any of the obstacles. You should print either this maximum radius r (rounded to two decimal places) or the message "impossible" if no such radius exists.

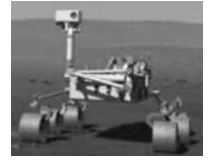
Sample Input

```
6 2
4 2 -1 4 -1 4 1 2 1
3 3 0 6 -1 6 1
8 2
3 1 -1 4 -1 4 4
3 3 -4 6 1 3 1
10 7
4 0 5 4 2 5 3 4 5
3 4 -5 9 -5 9 0
4 8 -5 11 -5 11 -2 8 -2
3 8 3 16 1 11 5
4 21 -5 23 -3 20 -2 15 -4
3 22 3 26 -1 28 0
3 24 0 29 4 25 3
0 0
```

Sample Output

```
1.00
impossible
1.33
```





Obstacle Course

You are working on the team assisting with programming for the Mars rover. To conserve energy, the rover needs to find optimal paths across the rugged terrain to get from its starting location to its final location. The following is the first approximation for the problem.

$N \times N$ square matrices contain the expenses for traversing each individual cell. For each of them, your task is to find the minimum-cost traversal from the top left cell $[0][0]$ to the bottom right cell $[N-1][N-1]$. Legal moves are up, down, left, and right; that is, either the row index changes by one or the column index changes by one, but not both.

Input

Each problem is specified by a single integer between 2 and 125 giving the number of rows and columns in the $N \times N$ square matrix. The file is terminated by the case $N = 0$.

Following the specification of N you will find N lines, each containing N numbers. These numbers will be given as single digits, zero through nine, separated by single blanks.

Output

Each problem set will be numbered (beginning at one) and will generate a single line giving the problem set and the expense of the minimum-cost path from the top left to the bottom right corner, exactly as shown in the sample output (with only a single space after "Problem" and after the colon).

Sample Input

```
3
5 5 4
3 9 1
3 2 7
5
3 7 2 0 1
2 8 0 9 1
1 2 1 8 1
9 8 9 2 0
3 6 5 1 5
7
9 0 5 1 1 5 3
4 1 2 1 6 5 3
0 7 6 1 6 8 5
1 1 7 8 3 2 3
9 4 0 7 6 4 1
5 8 3 2 4 8 3
7 4 8 4 8 3 4
0
```

Sample output

```
Problem 1: 20
Problem 2: 19
Problem 3: 36
```



Pencils from the 19th Century

Before “automaton” was a theoretic computer science concept, it meant “mechanical figure or contrivance constructed to act as if by its own motive power; robot.” Examples include fortunetellers, as shown above, but could also describe a pencil seller, moving pencils from several baskets to a delivery trough.

On National Public Radio, the Sunday Weekend Edition program has a “Sunday Puzzle” segment. The show that aired on Sunday, 29 June 2008, had the following puzzle for listeners to respond to (by Thursday, 3 July, at noon through the NPR web site):

From a 19th century trade card advertising Bassetts Horehound Troches, a remedy for coughs and colds: A man buys 20 pencils for 20 cents and gets three kinds of pencils in return. Some of the pencils cost four cents each, some are two for a penny and the rest are four for a penny. How many pencils of each type does the man get?

One clarification from the program of 6 July: correct solutions contain at least *one* of each pencil type.

For our purposes, we will expand on the problem, rather than just getting 20 pencils for 20 cents (which is shown in the sample output below). The input file will present a number of cases. For each case, give all solutions or print the text “No solution found”. Solutions are to be ordered by increasing numbers of four-cent pencils.

Input

Each line gives a value for **N** ($2 \leq \mathbf{N} \leq 256$), and your program is to end when **N**=0 (at most 32 problems).

Output

The first line gives the instance, starting from 1, followed by a line giving the statement of the problem. Solutions are shown in the three-line format below followed by a blank line, or the single line “No solution found”, followed by a blank line. Note that by the nature of the problem, once the number of four-cent pencils is determined, the numbers of half-cent and quarter-cent pencils are also determined.

```
Case n:
nn pencils for nn cents
nn at four cents each
nn at two for a penny
nn at four for a penny
```

Sample Input

10
20
40
0

Sample Output

Case 1:
10 pencils for 10 cents
No solution found.

Case 2:
20 pencils for 20 cents
3 at four cents each
15 at two for a penny
2 at four for a penny

Case 3:
40 pencils for 40 cents
6 at four cents each
30 at two for a penny
4 at four for a penny

7 at four cents each
15 at two for a penny
18 at four for a penny

Optimal Strategy for the ICPC

Thanks in large part to the success of Mr. Data as an officer in Star Fleet, the International Collegiate Programming Contest, in a striking move towards android rights, has established a contest open to androids teams. Biological entities are allowed to compete, if they dare.



The following is taken from the official rules for the International Collegiate Programming Contest, as reported at <http://www.acmicpc-pacnw.org/rules.htm>

The total time is the sum of the time consumed for each problem solved. The time consumed for a solved problem is the time elapsed from the beginning of the contest to the submittal of the first accepted run plus 20 penalty minutes for every previously rejected run for that problem. There is no time consumed for a problem that is not solved.

Quite simply, one element of the optimal strategy is *not* to have any erroneous submissions, so the androids do not have to worry about the penalty minutes. All that remains is to determine the order in which they should submit problems.

Let's assume perfect knowledge — hey, these androids are *good* — so that they can make a very good estimate of the development time required for each of the problems. The task is to determine the *order* in which the problems should be submitted. The androids realize that their best approach is for each to think independently about different problems rather than having all three work on a single problem. Furthermore, each android types infinitely fast, and does not use the computer terminal while thinking. Hence, up to three problems can be simultaneously in progress at any given time, and it is actually possible for all three bots to submit a problem within the same minute. For the same reason, the number of problems posed is larger than those posed in the contest for biological entities. Being innately fussy, if there are multiple ways to submit the problems and obtain the same score, they will submit the problem order that comes lexicographically first.

Determine the algorithm to solve the most problems and to obtain the best possible score for those problems. Then implement it.

Input

The first line of input to your program is a single integer n ($0 < n < 100$), giving the number of data sets — one for each set of problems. Following that are exactly n lines, giving information about each data set. The first number is the number of problems in that dataset as an integer k ($5 \leq k \leq 15$). On the same line, separated by single spaces, are k integers, all between 1 and 300 inclusive, giving the estimated time required to solve each problem. The problems themselves are labeled by alphabetic characters starting with A. Note that there are exactly 300 minutes in the contest.

Output

Each data set generates one line of output, giving the data set number, the sequence the problems are submitted, the total number solved, and the final penalty score. See the sample output for format — all non-blank entries are separated by single blank spaces.

Sample input

```
4
9 25 50 100 150 100 100 150 225 300
10 60 120 99 129 15 150 225 135 50 123
12 6 60 99 45 135 66 231 63 96 39 50 123
15 75 75 75 75 75 75 75 75 75 75 75 75 75 75 75
```

Sample output

```
Data set 1: A B C D E F G H 8 1450
Data set 2: E I A J C B F H D 9 1473
Data set 3: A J D B K F H I C E L 11 1452
Data set 4: A B C D E F G H I J K L 12 2250
```

Symbolic Logic Mechanization

Marvin, the robot with a brain the size of a planet, followed some . . . markedly less successful robots as the product line developed. One such was Monroe, the robot — except, to help him recognize his name, he was referred to as Moe. He is sufficiently mentally challenged that he needs external assistance to handle symbolic logic.



Polish notation is the prefix symbolic logic notation developed by Jan Łukasiewicz (1929). [Hence postfix expressions are referred to as being in Reverse Polish Notation — RPN.] The notation developed by Łukasiewicz (referred to as PN below) uses upper-case letters for the logic operators and lower-case letters for logic variables (which can only be **true** or **false**). Since prefix notation is self-grouping, there is no need for precedence, associativity, or parentheses, unlike infix notation. In the following table the PN operator is shown, followed by its operation. Operators not having exactly equivalent C/C++/Java operators are shown in the truth table (using 1 for **true** and 0 for **false**). [The operator J is not found in Łukasiewicz’ original work but is included from A.N.Prior’s treatment.]

PN Operator	Operation
Cpq	conditional
Np	not
Kpq	and
Apq	(inclusive) or
Dpq	nand
Epq	equivalence
Jpq	exclusive or

Truth Tables				
p	q	Cpq	Dpq	Epq
0	0	1	1	1
0	1	1	1	0
1	0	0	1	0
1	1	1	0	1

For every combination of PN operators and variables, an expression is a “well-formed formula” (WFF) if and only if it is a variable or it is a PN operator followed by the requisite number of operands (WFF instances). A combination of symbols will fail to be a “well-formed formula” if it is composed of a WFF followed by extraneous text, it uses an unrecognized character [upper-case character not in the above table or a non-alphabetic character], or it has insufficient operands for its operators. For invalid expressions, report the *first* error discovered in a left-to-right scan of the expression. For instance, immediately report an error on an invalid character. If a valid WFF is followed by extraneous text, report that as the error, even if the extraneous text has an invalid character.

In addition, every WFF can be categorized as a tautology (true for all possible variable values), a contradiction (false for all possible variable values), or a contingent expression (true for some variable values, false for other variable values).

The simplest contingent expression is simply “p”, true when p is true, false when p is false. One very simple contradiction is “KpNp”, both p and not-p are true. Similarly, one very simple

tautology is “ $\neg p \vee p$ ”, either p is true or $\neg p$ is true. For a more complex tautology, one expression of De Morgan’s Law is “ $\neg(\neg p \wedge \neg q)$ ”.

Input

Your program is to accept lines until it receives an empty character string. Each line will contain only alphanumeric characters (no spaces or punctuation) that are to be parsed as potential “WFFs”. Each line will contain fewer than 256 characters and will use at most 10 variables. There will be at most 32 non-blank lines before the terminating blank line.

Output

For each line read in, echo it back, followed by its correctness as a WFF, followed (if a WFF) with its category (tautology, contradiction, or contingent). In processing an input line, immediately terminate and report the line as not a WFF if you encounter an unrecognized operator (even though it may fail to be well-formed in another way as well). If it has extraneous text following the WFF, report it as incorrect. If it has insufficient operands, report that. Use *exactly* the format shown in the Sample Output below.

Sample Input

```
q
Cp
Cpq
A01
Cpqr
ANpp
KNpp
Qad
CKNppq
JDpqANpNq
CDpwANpNq
EDpqANpNq
KCDpqANpNqCANpNqDpq
[this is an empty line]
```

Sample Output

```
q is valid: contingent
Cp is invalid: insufficient operands
Cpq is valid: contingent
A01 is invalid: invalid character
Cpqr is invalid: extraneous text
ANpp is valid: tautology
KNpp is valid: contradiction
Qad is invalid: invalid character
CKNppq is valid: tautology
JDpqANpNq is valid: contradiction
CDpwANpNq is valid: contingent
EDpqANpNq is valid: tautology
KCDpqANpNqCANpNqDpq is valid: tautology
```

Repair Depots

RoboCorp Oregon has already deployed several of its PoliceBots throughout the state, and suddenly they have remembered the importance of being able to repair these bots when they break. They are willing to build only a limited number of repair depots in the state, and they want to locate these depots so that each PoliceBot's city is within a given distance of some depot.

You are given the locations of where the n ($1 \leq n \leq 16$) PoliceBots have been deployed, and the maximum number ($1 \leq c \leq n$) of repair depots that RoboCorp is willing to construct. Your job is to locate these repair depots to minimize the distance any bot has to be transported for repair, and return what that maximum distance is. You can assume that any bot can be repaired at any depot.

Input

The first line contains the integer t ($1 \leq t \leq 350$), the number of cases. Each case starts with a line containing n and c . Following this line is one line per deployed bot; that line contains two floating point values, separated by a single space, giving the Cartesian coordinates of the bot. Each such value will be between 0.0 and 10.0, inclusive (NOTE: the Sample Input shows integer values for brevity).

Output

You are to print for each input set the minimum value that can be obtained, such that each PoliceBot is within that distance of some depot. You should print your result with at least one digit before and exactly six digits after the decimal point; your value should be within $5e-7$ of the true result. To make round-off error less of a concern, the true result will never have a 4 or a 5 as the seventh digit after the decimal point.

Sample input

```
2
9 3
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
4 2
0 0
1 1
2 4
3 9
```

Sample output

```
1.000000
2.236068
```



Crazy Circuits

You've just built a circuit board for your new robot, and now you need to power it. Your robot circuit consists of a number of electrical components that each require a certain amount of current to operate. Every component has a + and a - lead, which are connected on the circuit board at junctions. Current flows through the component from + to - (but note that a component does not "use up" the current: everything that comes in through the + end goes out the - end).

The junctions on the board are labeled $1, \dots, N$, except for two special junctions labeled + and - where the power supply terminals are connected. The + terminal only connects + leads, and the - terminal only connects - leads. All current that enters a junction from the - leads of connected components exits through connected + leads, but you are able to control how much current flows to each connected + lead at every junction (though methods for doing so are beyond the scope of this problem¹). Moreover, you know you have assembled the circuit in such a way that there are no feedback loops (components chained in a manner that allows current to flow in a loop).

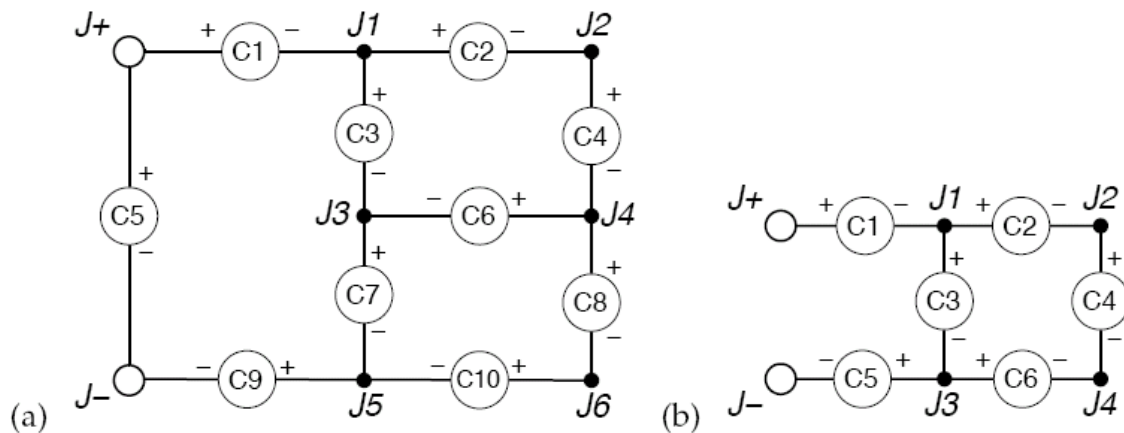


Figure 1: Examples of two valid circuit diagrams. In (a), all components can be powered along directed paths from the positive terminal to the negative terminal. In (b), components 4 and 6 cannot be powered, since there is no directed path from junction 4 to the negative terminal.

In the interest of saving power, and also to ensure that your circuit does not overheat, you would like to use as little current as possible to get your robot to work. What is the smallest amount of current that you need to put through the + terminal (which you can imagine all necessarily leaving through the - terminal) so that every component on your robot receives its required supply of current to function?

¹ For those who are electronics-inclined, imagine that you have the ability to adjust the potential on any component without altering its current requirement, or equivalently that there is an accurate variable potentiometer connected in series with each component that you can adjust. Your power supply will have ample potential for the circuit.

Input

The input file will contain multiple test cases. Each test case begins with a single line containing two integers: \mathbf{N} ($0 \leq \mathbf{N} \leq 50$), the number of junctions not including the positive and negative terminals, and \mathbf{M} ($1 \leq \mathbf{M} \leq 200$), the number of components in the circuit diagram. The next \mathbf{M} lines each contain a description of some component in the diagram. The i^{th} component description contains three fields: \mathbf{p}_i , the positive junction to which the component is connected, \mathbf{n}_i , the negative junction to which the component is connected, and an integer \mathbf{I}_i ($1 \leq \mathbf{I}_i \leq 100$), the minimum amount of current required for component i to function. The junctions \mathbf{p}_i and \mathbf{n}_i are specified as either the character '+' indicating the positive terminal, the character '-' indicating the negative terminal, or an integer (between 1 and \mathbf{N}) indicating one of the numbered junctions. No two components have the same positive junction and the same negative junction. The end-of-file is denoted by an invalid test case with $\mathbf{N} = \mathbf{M} = 0$ and should not be processed.

Output

For each input test case, your program should print out either a single integer indicating the minimum amount of current that must be supplied at the positive terminal in order to ensure that every component is powered, or the message "impossible" if there is no way to direct a sufficient amount of current to each component simultaneously.

Sample Input

```
6 10
+ 1 1
1 2 1
1 3 2
2 4 5
+ - 1
4 3 2
3 5 5
4 6 2
5 - 1
6 5 3
4 6
+ 1 8
1 2 4
1 3 5
2 4 6
3 - 1
3 4 3
0 0
```

Sample Output

```
9
impossible
```

PropBot

You have been selected to write the navigation module for PropBot. Unfortunately, the mechanical engineers have not provided a lot of flexibility in movement; indeed, the PropBot can only make two distinct movements. It can either move 10 cm forward, or turn towards the right by 45 degrees. Each of these individual movements takes one second of time.

Input

Your module has two inputs: the Cartesian coordinates of a point on the plane that the PropBot wants to get as close to as possible, and the maximum number of seconds that can be used to do this. At the beginning of the navigation, the robot is located at the origin, pointed in the $+x$ direction.

The number of seconds will be an integer between 0 and 24, inclusive. Both the x and y coordinates of the desired destination point will be a real number between -100 and 100, inclusive.

The first entry in the input file will be the number of test cases, t ($0 < t \leq 100$). Following this line will be t lines, with each line containing three entries separated by spaces. The first entry will be the number of seconds PropBot has to get close to the point. The second entry is the x -coordinate of the point, and the third entry is the y -coordinate of the point.

Output

Your program must return the distance between the goal point and the closest point the robot can get to within the given time.

Your result should include at least one digit to the left of the decimal point, and exactly six digits to the right of the decimal point. To eliminate the chance of round off error affecting the results, we have constructed the test data so the seventh digit to the right of the decimal point of the true result is never a 4 or a 5.

Sample Input

```
2
24 5.0 5.0
9 7.0 17.0
```

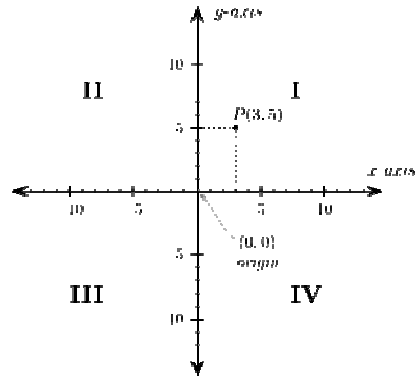
Sample Output

```
0.502525
0.100505
```


Problem J

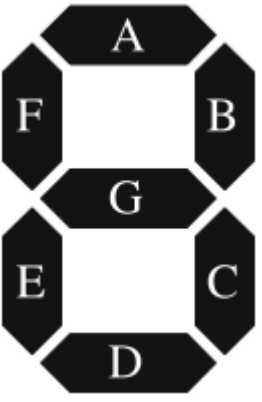

Input: j.in

Output: stdout/cout/System.out



“Ancient” Calculator

While travelling through the wilderness on Earth (and discussing the concept of a ‘Zeroeth’ Robotic Law), R. Daneel Olivaw and R. Giskard Reventlov happened upon a small device protruding from the ground. After some initial analysis, the two robots determine it is an ancient calculating device. More specifically, it appears to be a four function calculator. Unfortunately the display is covered with centuries of debris which appears difficult to remove. Amazingly, the calculator still functions, but of course you can’t see the results. This calculator has a display constructed from seven-segment LEDs, as shown here:

Seven-Segment display	The decimal numbers
	

This particular calculator only displays three digits of accuracy. There are three seven segment digits, and there are no decimal points since all math is performed using integers (with truncation where appropriate).

If the number on the display happens to be negative, there is a minus sign in front of the number. For a number such as -6, this is done by making the right-most digit a “6” and using the middle segment of the next digit over as the minus sign. For a number such as -123, there is one additional segment to the left of the display. Thus, the number -112 lights up 10 segments, made up of 9 for the number, and one more for the minus sign. The display thus ranges from “-999” to “999”. There is no “plus sign” for positive numbers.

In order to prove that the calculator still operates, and justify scraping off the paint, the back cover is removed and an ammeter is attached to the display. An ammeter measures the current consumed by the device, and each segment of a digit consumes 5 milliamps per segment. Thus, to display a number such as “798” the current consumption can be predicted as follows:

```
Digit 7 - 3 segments lit = 15 milliamps
Digit 9 - 6 segments lit = 30 milliamps
Digit 8 - 7 segments lit = 35 milliamps
Total = 80 milliamps
```

Of course, it’s apparent that the number “897” also consumes the same amount of current, as does “789” or, for that matter, “-891”.

The calculator allows you to enter numbers with an optional minus sign and up to three digits, and to either add, subtract, multiply, or divide the numbers. Daneel wishes to prove that the functionality of the calculator is intact, so he enters “949” and measures 80 milliamps. Then he pushes the subtract key. Next he enters “51” and measures 35 milliamps. After pressing the “=” key the result should be “898”, measuring 100 milliamps. As a second example, he enters “-5”, then pushes the addition key. He then enters “-4” and presses equal. The answer is “-9”; the measurements were 30 milliamps, 25 milliamps, and 35 milliamps respectively.

Giskard is unsure of this result, so he presents Daneel with the following problem: given the current consumption (milliamps) of operand X, operand Y, and result Z, and given unknown operation Op, determine all possible values for $X \text{ Op } Y = Z$, assuming that possible solutions exist. If no such solutions exist, report it as shown in the **Sample Output** section.

The calculator has only three digits on the display, and neither Giskard nor Daneel are sure what it will do if the result of an expression is outside the range of -999 to 999. Thus, they will not consider such expressions as possible solutions.

Example

An example is as shown above; for inputs 80, 35, 100 one result is “925 – 117 = 808”. Some input combinations have no result. For the inputs 35, 10, 10 there is no expression that matches those readings. For any given input set, the program should determine all of the valid expressions and print the number of expressions found.

Input

There will be multiple cases. Each input line except the last contains three integer values between 0 and 999, inclusive, each separated by a space, which are the X, Y, and Z values for a single case. Each number is in milliamps according to the above description. The last line will contain the single value 0, indicating the end of the problem set.

Output

Note that no leading zeros should be considered – the number “12” for example, is never entered on the calculator as “012” even though the result of the calculation may be the same. Thus, no expression should contain numbers with leading zeros, either as input or as result. For each case, find all possible expressions and report how many there are in a grammatically correct form, as shown below, as well as echoing back the input.

Sample Input

```
10 10 5
10 20 30
10 30 20
30 65 60
30 65 65
30 95 95
35 35 110
0
```

Sample Output

```
0 solutions for 10 10 5
4 solutions for 10 20 30
1 solution for 10 30 20
516 solutions for 30 65 60
819 solutions for 30 65 65
304 solutions for 30 95 95
2 solutions for 35 35 110
```