2004 Mid-Atlantic Regional Programming Contest

Welcome to the 2004 Programming Contest. Before you start the contest, please be aware of the following notes:

1. There are eight (8) problems in the packet, using letters A–H. These problems are NOT sorted by difficulty. As a team's solution is judged correct, the team will be awarded a balloon. The balloon colors are as follows:

Problem	Problem Name	Balloon Color
A	Filling Out the Team	Blue
B	Auctions R Us	Purple
C	GHOST	Pink
D	Doggone Moles	Green
E	Line of Sight	Gold
F	Tangled in Cables	Orange
G	All Roads Lead to Albuquerque, er, Rome	Red
H	Balanced Budget Initiative	Silver

2. During the contest, a variety of reference materials for the supported compilers, libraries, and editors are available at

   `http://midatl.cs.vt.edu/contest/`

   The contest scoreboard is also accessible through that URL. Note that during the contest you may only access web pages through that link.

3. All solutions must read from standard input and write to standard output. In C this is scanf/printf, in C++ this is cin/cout, and in Java this is System.in/System.out. The judges will ignore all output sent to standard error. (You may wish to use standard error to output debugging information.) From your workstation you may test your program with an input file by redirecting input from a file:

   ```
   program < file.in
   ```

4. Solutions for problems submitted for judging are called runs. Each run will be judged. Runs for each particular problem will be judged in the order they are received. However, it is possible that runs for different problems may be judged out of order. For example, you may submit a run for B followed by a run for C, but receive the response for C first. DO NOT request clarifications on when a response will be returned. If you have not received a response for a run within 60 minutes of submitting it, **you may have a runner ask the site judge to determine the cause of the delay. Under no circumstances should you ever submit a clarification request about a submission for which you have not received a judgment.**

   The judges will respond to your submission with one of the following responses. In the event that more than one response is applicable, the judges may respond with any of the applicable responses.

Response	Explanation
Correct	Your submission has been judged correct.
Incorrect Output	Your submission generated output that is not correct.
Output Format Error	Your submission's output is not in the correct format or is misspelled.
Incomplete Output	Your submission did not produce all of the required output.
Excessive Output	Your submission generated output in addition to or instead of what is required.
Compilation Error	Your submission failed to compile.
Run-Time Error	Your submission experienced a run-time error.
Time-Limit Exceeded	Your submission did not solve the judges' test data within 30 seconds.

5. A team's score is based on the number of problems they solve and penalty points, which reflect the amount of time and number of incorrect submissions made before the problem is solved. For each problem solved correctly, penalty points are charged equal to the time at which the problem was solved plus 20 minutes for each incorrect submission. No penalty points are added for problems that are never solved. Teams are ranked first by the number of problems solved and then by the fewest penalty points.

6. This problem set contains sample input and output for each problem. However, you may be assured that the judges will test your submission against several other more complex datasets, which will not be revealed until after the contest. Your major challenge is designing other input sets for yourself so that you may fully test your program before submitting your run. Should you receive an incorrect judgment, you should consider what other datasets you could design to further evaluate your program.

7. In the event that you feel a problem statement is ambiguous, you may request a clarification. Read the problem carefully before requesting a clarification. If the judges believe that the problem statement is sufficiently clear, you will receive the response, "The problem statement is sufficient; no clarification is necessary." If you receive this response, you should read the problem description more carefully. If you still feel there is an ambiguity, you will have to be more specific or descriptive of the ambiguity you have found. If the problem statement is ambiguous in specifying the correct output for a particular input, please include that input data in the clarification request.

   Additionally, you may submit a clarification request asking for the correct output for input you provide. The judges will seek to respond to these requests with the correct output. These clarification requests will be answered only when no clarifications regarding ambiguity are pending. The judges reserve the right to suspend responding to these requests during the contest. If the provided input does not meet the specifications of the problem, or contains numbers that are not representable using the available programming languages, the response "illegal input" will be returned.

   If a clarification, including output for a given input, is issued during the contest, it will be broadcast to all teams.

8. The submission of abusive programs or clarification requests to the judges will be considered grounds for immediate disqualification.

9. All lines end with a newline character (`\n`, `endl`, or `println()`).

10. All input sets used by the judges will follow the input format specification found in the problem description.

11. Unless otherwise specified, all numbers will appear in the input and should be presented in the output beginning with the `-` if negative, followed immediately by 1 or more decimal digits. If it is a real number, then the decimal point appears, followed by any number of decimal digits (for output of real numbers the number of decimal digits will be specified in the problem description as the "precision"). All real numbers printed to a given precision should be rounded to the nearest value.

    In simpler terms, neither scientific notation nor commas will be used for numbers, and you should ensure you use a printing technique that rounds to the appropriate precision.

12. Good luck, and HAVE FUN!!!

Problem A: Filling Out the Team

Over the years, the people of the great city of Pittsburgh have repeatedly demonstrated a collective expertise at football second to none. Recently a spy has discovered the true source of the city's football power—a wizard known only as "Myron," who is infallible at selecting the proper position at which each player will excel.

Now that you know the source of Pittsburgh's wisdom, you are determined to provide your school's football team with a computer program that matches the wisdom of "Myron." You have consulted with the best football minds you can find, and they have dispensed their wisdom on the slowest speed, minimum weight, and minimum strength required to play each position.

Position	Slow. Speed	Min. Weight	Min. Strength
Wide Receiver	4.5	150	200
Lineman	6.0	300	500
Quarterback	5.0	200	300

Using this knowledge, you will develop a program that reads in several players physical attributes and outputs what position(s) they are able to play.

Input

Each line of the input file will list the attributes for one player:

```
<speed> <weight> <strength>
```

Each number will be a real-valued number. The file will end with a line reading "0 0 0"

Output

For each player, you will output one line listing the positions that player can play. A player can play a position if each of their attributes is greater or equal to the minimum for weight and strength, and less than or equal to the slowest speed. If a player can play multiple positions, output them in the order listed above, separated by whitespace. You may leave an extra space at the end of the line. If a player can play no positions, write "No positions" on the line.

Example

Input:

```
4.4 180 200
5.5 350 700
4.4 205 350
5.2 210 500
0 0 0
```

Output:

```
Wide Receiver
Lineman
Wide Receiver Quarterback
No positions
```

Problem B: Auctions R Us

Having run into trouble with current online auctions and buyers that win auctions and then back out, you decide to open a new enterprise that has the bidders deposit funds before they may bid on any item. If they win an auction, the amount they bid is immediately (that second!) deducted from their account. (The problem of sellers that don't deliver the items will be left for another day.)

You must write a program to implement the rules of this auction. You will be auctioning off a number of items, each of which will have a reserve price that must be met. Each of your bidders will deposit funds with you, and you must match these funds with items they successfully bid for. You will write a program that tracks the auctions during a single day and outputs the results of each auction.

Auction Rules

You are guaranteed:

- No two items will have the same end time.
- No two bids will have the same bid time.
- No price, bid, or account balance will be negative.

Bidder numbers and item numbers are unique within each category, but a bidder may have the same number as an item. Bidder and item numbers are not necessarily assigned sequentially.

An auction is won by the highest bid that:

- arrives no later than the second the auction ends.
- is greater than or equal to the minimum price for the item
- has at least the bid amount remaining in the bidder's account at the instant the auction ends.

Input

There are 3 sections in the data file, describing the items available for bid, the registered bidders, and the bids made during the auction.

Items

- A single line containing the number of items, i
- i lines, one for each item of the form:

  `<item number> <minimum price> <auction end time>`

  Item number is a non-negative integer, *minimum price* is specified to the penny (0.01), and *auction end time* is in 24 hour format of the form XX:YY:ZZ where XX is in hours from 00 to 23, YY is in minutes from 00 to 59, and ZZ is in seconds from 00 to 59.

Bidders

- A single line with the number of bidders registered, j
- j lines of bidder data of the form:

  `<bidder number> <account balance>`

  Where *bidder number* is a non-negative integer and *account balance* is specified to the penny (0.01).

Bids

- A single line with the number of bids received, k
- k lines of bid data of the form:
  `<item # being bid on> <bidder number> <bid amount> <bid time>`
  where all fields are formatted as described above.

Output

Output one line for each item being auctioned, in order of their auction finish time, listing

`Item <item number> Bidder <bidder number> Price <winning bid>`

If there is not a winning bid for an item, for that item output

`Item <item number> Reserve not met.`

Example

Input:

```
2
1 5.00 05:06:27
2 25.00 15:30:11
2
11 37.37
22 55.55
3
1 11 60.00 04:03:01
2 11 26.00 00:18:03
2 22 27.00 09:03:05
```

Output:

```
Item 1 Reserve not met.
Item 2 Bidder 22 Price 27.00
```

Problem C: GHOST

GHOST is a spelling game played by bored school kids on long car/bus rides. The purpose of the game it to accumulate letters that spell some word without ever actually finishing a word. Before the game begins, players agree on the order in which they will play. Plays proceed from one player to the next, returning then to the first player until the game is completed. Each player must, in turn, 1) extend the current "word", 2) bluff, or 3) challenge.

1. The most common play is to extend the current sequence of letters by adding a single letter, so that the resulting sequence of letters forms the beginning of some word. For example, the first player might call "P", thinking (secretly) of the word "part". The second player might call "L", thinking of the word "play". The third player might call "E", thinking of the word "please".

   A player loses if they actually complete a valid English word of 4 or more letters. For example, if there were only three players, after "PLE" the first player might try to extend the word by calling "A", thinking of the word "plead". This would, however, be a losing play because "plea" is a valid word.

2. A player who cannot think of a valid letter to extend the current sequence may opt to "bluff" by calling out an arbitrary letter, hoping that the next player will not notice.

3. Finally, if a player believes, on his turn, that the preceding player was bluffing or that the preceding player completed a word, he may challenge the preceding player. If all players agree that the current sequence completes a word (of at least 4 letters), the preceding player loses. If the preceding player cannot name a word that can be formed from the current sequence, the preceding player loses. If the current sequence is not a valid word and the preceding player is able to name a possible word beginning with the sequence, the challenger loses.

Write a program to serve as a player in a game of GHOST. Note that a skillful player will, on her turn, not only worry about coming up with a legal extension to the current sequence of letters, but will also think about all the words that could be formed from an extension and whether, comparing the number of letters in those words to the number of players, consider whether a possible extension could result in her getting stuck on a future turn with no legal extension that does not end a word, thus losing the game.

Input

The input for this game will consist of a sequence of one or more scenarios.

Each scenario contains the following:

The first line of the scenario will contain a single integer indicating the number of players in the game. This value will be at least 2 for a valid scenario. The end of the input file will be indicated by a value less than 2 for this number.

Following this will be a list of words to serve as the program's dictionary/vocabulary for the scenario. Each word will appear on a separate line, with no leading, trailing, or internal whitespace. Each word will consist only of the characters {a–z}. The end of this list of words will be signaled by an empty line.

Following that empty line, the final input line of the scenario will contain the current sequence of letters, again with no leading or trailing spaces. This sequence may be empty if the computer player is the first player. The sequence may also contain more letters than the number of players, indicating that all players (including the computer player) have taken one or more turns.

Output

Your program will produce a single line of output for each scenario.

That line of output will consist of the current sequence of letters from the input, followed by a single blank, followed by:

1. The word "Challenge" if the current sequence is a complete word in the vocabulary list or is not a prefix of any word in the vocabulary list, or

2. A single character representing a valid extension if it can find some word formed with that extension such that this extension does not complete a word and, if all other players continue to spell out that same word, neither that word nor any shorter word would be completed on the computer player's turn.

   If multiple such extensions are possible, and if any extensions guarantee a different player will lose, the program should select one of those.

3. The word "Bluff" if the only possible extensions would lead to a loss by the computer player.

Example

Input:

```
3
area
arched
apple
apply
applied

ar
2
area
arch
apple
apply
applied
applying

a
2
area

ax
0
```

Output:

```
ar e
a p
ax Challenge
```

Problem D: Doggone Moles

A mole has pockmarked our yard with a rectangular grid of tunnels. Infuriated at the damage, we have released a number of terriers into the yard to catch the mole. The terriers have very sensitive hearing and, if they come close enough to the mole, can dig very quickly and catch the rodent. Unfortunately, the mole is very sensitive to the vibrations caused by the footsteps of the terriers, and will actively try to evade them.

We have no idea where the mole was when the terriers were released. But we have watched the terriers move about the yard for some time, and the mole has not been caught. Write a program that deduces where the mole might be, given our observations.

At the time we began recording our observations, we also know that the mole was not in a position underneath or adjacent to a terrier. In each subsequent time interval, the terriers may have remained in the same position or may have moved one space horizontally or vertically. Then the mole may have done the same. If, before or after any of these moves, by terriers or by the mole, a terrier were directly over the mole or in a position adjacent (horizontally or vertically) to the mole, the mole would have been caught.

Write a program that accepts a description of the yard and of the location of the terriers within it over a period of time. The program should print a list of the possible positions of the mole at the end of that time.

Input

The input for this program consists of one or more observation sets.

Each observation set is constructed as follows:

- The first line contains 4 integers

  `<W> <L> <N> <T>`

  W and L are positive integers representing the width (x dimension) and length (y dimension) of the yard. N is the non-negative number of terriers. T is the positive number of time intervals over which we have conducted observations.
- The remainder of the observation set contains one line per terrier. Each line contains $2T$ integers denoting the (x,y) coordinates of the terrier at each of the T time steps, expressed separated by whitespace without parentheses or commas. Possible coordinates range from $(0, 0)$ in one corner of the yard to (W, L) at the opposite corner.

The end of input is signaled by a line containing 4 zeros in place of a valid (W, L, N, T) set.

Output

For each observation set, your program should print a line "Observation Set " followed by the integer number of the set (starting at 1).

If there is at least one possible location for the mole, then, beginning on the next line, print all the possible locations of the mole as (x, y) pairs, 8 pairs per output line (except possibly fewer for the final line of output for the set). There should be no leading blanks before the first pair on a line nor trailing blanks after the final pair on the line, but successive pairs on the same line should be separated by exactly one blank. A pair is printed in the format "(x, y)" with no internal blanks. Pairs should be printed in an order such that pairs with lower values of y come before any pairs

with higher values of y and, for pairs with the same y value, pairs with lower values of x come before pairs with higher values of x.

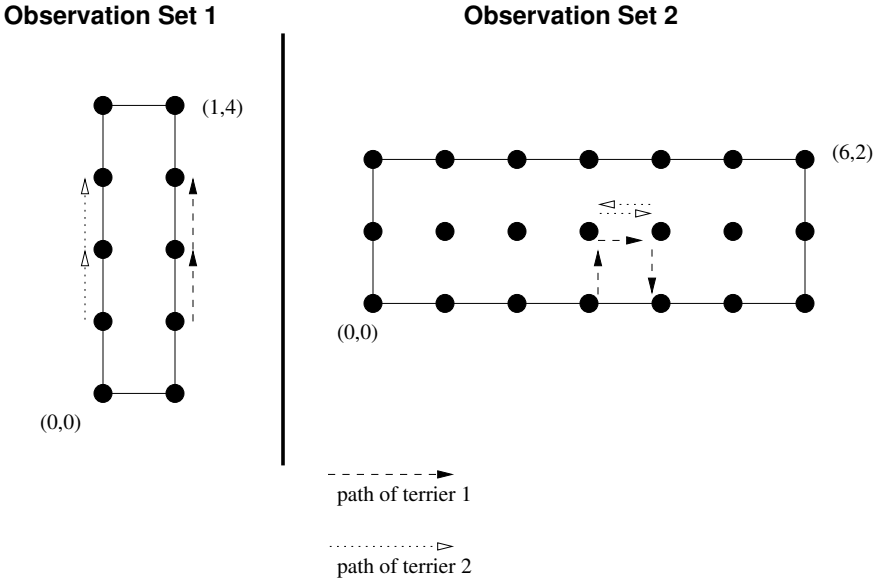
If there are no possible locations for the mole, then the second line of output for the observation set will consist of the message "No possible locations".

Example

Input:

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

This input set may be visualized as:

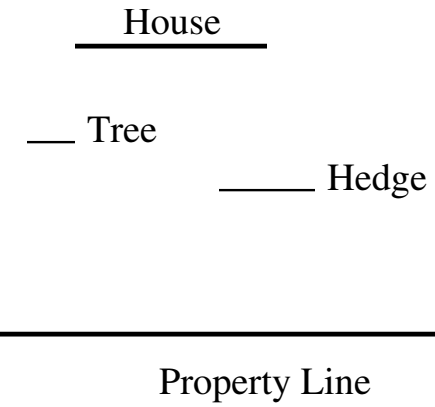


Output:

```
Observation Set 1
No possible locations
Observation Set 2
(0,0) (1,0) (2,0) (6,0) (0,1) (1,1) (5,1) (6,1)
(0,2) (1,2) (2,2) (4,2) (5,2) (6,2)
```

Problem E: Line of Sight

An architect is very proud of his new home and wants to be sure it can be seen by people passing by his property line along the street. The property contains various trees, shrubs, hedges, and other obstructions that may block the view. For the purpose of this problem, model the house, property line, and obstructions as straight lines parallel to the x axis:



To satisfy the architect's need to know how visible the house is, you must write a program that accepts as input the locations of the house, property line, and surrounding obstructions and calculates the longest continuous portion of the property line from which the entire house can be seen, with no part blocked by any obstruction.

Input

Because each object is a line, it is represented in the input file with a left and right x coordinate followed by a single y coordinate:

```
<x1> <x2> <y>
```

Where $x1$, $x2$, and y are non-negative real numbers. $x1 < x2$

An input file can describe the architecture and landscape of multiple houses. For each house, the first line will have the coordinates of the house. The second line will contain the coordinates of the property line. The third line will have a single integer that represents the number of obstructions, and the following lines will have the coordinates of the obstructions, one per line.

Following the final house, a line "0 0 0" will end the file.

For each house, the house will be above the property line (*house y* $>$ *property line y*). No obstruction will overlap with the house or property line, e.g. if *obstacle y* $=$ *house y*, you are guaranteed the entire range *obstacle*$[x1, x2]$ does not intersect with *house*$[x1, x2]$.

Output

For each house, your program should print a line containing the length of the longest continuous segment of the property line from which the entire house can be to a precision of 2 decimal places. If there is no section of the property line where the entire house can be seen, print "No View".

Example

Input:

```
2 6 6
0 15 0
3
1 2 1
3 4 1
12 13 1
1 5 5
0 10 0
1
0 15 1
0 0 0
```

Output:

```
8.80
No View
```

Problem F: Tangled in Cables

You are the owner of SmallCableCo and have purchased the franchise rights for a small town. Unfortunately, you lack enough funds to start your business properly and are relying on parts you have found in an old warehouse you bought. Among your finds is a single spool of cable and a lot of connectors. You want to figure out whether you have enough cable to connect every house in town. You have a map of town with the distances for all the paths you may use to run your cable between the houses. You want to calculate the shortest length of cable you must have to connect all of the houses together.

Input

Only one town will be given in an input file.

- The first line gives the length of cable on the spool as a real number.
- The second line contains the number of houses, N
- The next N lines give the name of each house's owner. Each name consists of up to 20 characters $\{$a–z,A–Z,0–9$\}$ and contains no whitespace or punctuation.
- Next line: M, number of paths between houses
- next M lines in the form

  `<house name A> <house name B> <distance>`

  Where the two house names match two different names in the list above and the distance is a positive real number. There will not be two paths between the same pair of houses.

Output

The output will consist of a single line. If there is not enough cable to connect all of the houses in the town, output

```
Not enough cable
```

If there is enough cable, then output

```
Need <X> miles of cable
```

Print X to the nearest tenth of a mile (0.1).

Example

Input:

```
100.0
4
Jones
Smiths
Howards
Wangs
5
Jones Smiths 2.0
Jones Howards 4.2
Jones Wangs 6.7
Howards Wangs 4.0
Smiths Wangs 10.0
```

Output:

```
Need 10.2 miles of cable
```

Problem G: All Roads Lead to Albuquerque, er, Rome

A friend of mine has an unusual method of driving around the city, which he says helps reduce the number of routes he must memorize in order to not get lost. He picks two locations as hubs ($H1$ and $H2$), assigns all other locations to either $H1$ or $H2$, and then learns the shortest path from all locations to and from their associated hub. If he then wishes to travel from A to B, he goes from A to the hub associated with A, then to the hub associated with B (if B is associated with the other hub than A), then to B. My friend always travels to the hubs, even if that means that he visits his destination two or three times.

Your program should analyze a city (a set of nodes and edge lengths) and pick the best pair of hubs and assignment of nodes to hubs. The best configuration will be the configuration that minimizes the average distance of the trips between all pairs of nodes. If more than one configuration yields the lowest average, your program can output any of them.

Input

The input contains several test cases. The first line of the input file contains a single integer indicating the number of test cases.

The input for each test case starts with a single line

```
<n> <m>
```

$2 \leq n \leq 50$ and $1 \leq m \leq 1000$. n is the number of locations in the city and m is the number of road segments that directly connect two locations in the city. There may be more than one road segment between a pair of locations, and a road segment may start and end at the same location.

Each of the next m lines will describe the road segment between two locations and will contain three integers

```
<a> <b> <d>
```

$1 \leq a \leq n$, $1 \leq b \leq n$, and $1 \leq d \leq 1000$. a and b are locations that describe the ends of the road segment and d is the distance required to travel from a to b (or b to a) along the road segment. There are no one-way roads.

There will always exist a path between any two locations along the given road segments.

Output

For each test case, output an optimal choice of hubs and assignment of locations to hubs by outputting a line containing n integers, separated by spaces. If the i-th location is a hub, the i-th integer should be zero. If the i-th location is not a hub, the i-th integer should give the number of the i-th location's hub (1 to n inclusive).

Example

Input:

```
3
3 2
1 2 40
2 3 20
7 10
1 1 1
1 2 2
2 4 2
4 3 2
3 1 2
2 3 5
3 7 10
7 6 1
5 6 1
4 5 1
16 15
1 8 1
2 8 1
3 8 1
4 9 1
5 9 1
6 9 1
7 8 1
8 9 3
9 10 1
8 11 1
8 12 1
8 13 1
9 14 1
9 15 1
9 16 1
```

Output:

```
0 0 2
4 4 4 0 0 5 5
8 8 8 9 9 9 8 0 0 9 8 8 8 9 9 9
```

(for the first test case, `2 0 0` is also valid output)

Problem H: Balanced Budget Initiative

After bouncing 10 checks last month, you feel compelled to do something about your financial management. Your bank has started providing you with your statement online, and you believe that this is the opportunity to get your account in order by making sure you have the money to cover the checks you write.

Your bank provides you with a monthly statement that lists your starting balance, each transaction, and final balance. Your task is to compare the statement with the transactions from your checkbook register over the same time interval. You will identify transactions that appear in only the statement or register, as well as incorrect amounts recorded in the register (naturally the bank's statement is always correct) and math mistakes in your register.

Input

The bank statement appears first. It begins and ends with lines of the form:

```
balance <X>
```

with the first line indicating the starting balance and the second line indicating the final balance.

In between the balances is the list of transactions, one per line, in the form:

```
{check|deposit} <N> <X>
```

Where N is the integer check or deposit number (the same check or deposit number will only appear once, although the same number can apply to both a check and deposit), and X is the amount of the transaction.

Following the final balance the register entries appear. The first line of the register is the starting balance

```
<X>
```

Following are pairs of lines, with the next transaction appearing followed by the balance you calculated by hand after entering the transaction.

```
{check | deposit} <N> <X>
<X>
```

The pairs repeat until the end of the input file.

For all input numbers and intermediates, $|X| < 1000000$. All dollar amounts are given to the penny (0.01).

Output

For ease correcting your register, the output for each transaction occurs in the order it appears in the register. Each register entry receives exactly one line in the output.

If the register entry is entirely correct, meaning that it is found in the statement for the same amount, the math in the register is correct, and it is not a duplicate entry for a transaction previously found in the register, then output the line

```
{check|deposit} <N> is correct
```

However, if the transaction is not entirely correct, you will output a single line beginning with the transaction type and number, and one or more of the following mistakes, whitespace separated, in this order:

- **is not in statement** the transaction type and number do not occur in the statement
- **repeated transaction** the transaction has occurred previously in the register
- **incorrect amount** the register amount is different than the statement amount
- **math uses correct value** the math uses the value from the statement, although the actual transaction amount is recorded incorrectly in the register. This can only appear if **incorrect amount** is also displayed.
- **math mistake** the register balance after the transaction matches neither the statement amount for the transaction, nor the register entry for the transaction (if different than the statement amount)

Following the line for the final entry in the register, a listing of all transactions missing from the register will be printed. These items may be printed in any order, one per line:

```
missed {check|deposit} <N>
```

Example

Input:

```
balance 1000.00
check 100 10.00
check 101 20.00
check 102 30.00
check 103 100.00
deposit 1 10.00
deposit 2 20.00
deposit 3 30.00
deposit 4 500.00
balance 1400.00
1000.00
check 100 10.00
990.00
deposit 2 25.00
1015.00
check 101 20.00
990.00
check 102 30.00
960.00
check 101 21.00
940.00
check 103 100.00
840.00
deposit 3 30.00
870.00
deposit 4 500.00
1370.00
```

Output:

```
check 100 is correct
deposit 2 incorrect amount
check 101 math mistake
check 102 is correct
check 101 repeated transaction incorrect amount math uses correct value
check 103 is correct
deposit 3 is correct
deposit 4 is correct
missed deposit 1
```