Please do not open the problem packet until I tell you to do so.
This packet contains 9 problems lettered A-I.
Feel free to write in this packet. It is yours to keep.

Problem A: Problem A: Shuffling

Source: `shuffling.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

A casino owns an expensive card shuffling machine which may shuffle up to 520 cards at a time (there are 52 cards in each deck). For convenience, we will simply label the cards 1, 2, 3, ..., **N** where **N** is the total number of cards, and copies of the same card (e.g. Ace of Spades) from different decks are considered different. Unfortunately, the card shuffling machine is defective, and it always shuffles the cards the same way. The company that produces these machines is out of business because of the economic downturn. There is no one who can fix the machine, and a new machine is too expensive.

Being a brilliant employee of the casino, you realized that all is not lost. You can shuffle the cards differently simply by using the machine zero or more times. For example, suppose that the machine shuffles the cards 1, 2, 3, 4 into the order 2, 3, 4, 1. If you put the cards into the machine, take the shuffled cards out and insert them into the machine again (without changing the order), you will get the order 3, 4, 1, 2. That way, it is possible to shuffle the cards in many different ways even though it may take longer. But this is not a significant issue since decks do not have to be reshuffled often, and used decks can be shuffled while other decks are being used to avoid any waiting time.

Unfortunately, not all shufflings can be produced in this way in general, and you wish to know if this procedure "stack the decks" in a favorable way for the casino or the player. As a first step, you wish to know which shufflings are possible to produce, and how many times you need to use the machine on the deck in order to produce the shuffling.

Input

The input for each case consists of three lines. The first line consists of a single integer **N** indicating the number of cards to shuffle. The number of cards is a positive integer up to 520. The second line consists of the integers 1, 2, ..., **N** listed in some order and separated by a space. The list gives the order of the shuffling performed by the machine when the input cards are ordered 1, 2, ..., **N**. The third line is in the same format as the second line, and gives the shuffling we wish to obtain. The end of input is indicated by a line in which **N** = 0.

Output

For each case, print the smallest number of times (zero or more) you need to pass the deck through the machine to produce the desired shuffling. If it is not possible, print -1. The output for each case should be in a single line. You may assume that the answer will always fit in a 32-bit signed integer.

Sample Input

```
4
2 3 4 1
3 4 1 2
4
2 3 4 1
1 3 2 4
10
2 1 3 5 6 7 8 9 10 4
1 2 3 9 10 4 5 6 7 8
0
```

Sample Output

```
2
-1
12
```

Problem B: Coverage

Source: `coverage.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

A cell phone user is travelling along a line segment with end points having integer coordinates. In order for the user to have cell phone coverage, it must be within the transmission radius of some transmission tower. As the user travels along the path, cell phone coverage may be gained (or lost) as the user moves inside the radius of some tower (or outside of the radii of all towers). Given the location of up to 100 towers and their transmission radii, you are to compute the percentage of cell phone coverage the user has along the specified path. The (x,y) coordinates are integers between -100 and 100, inclusive, and the tower radii are integers between 1 and 100, inclusive.

Input

Your program will be given a sequence of configurations, one per line, of the form:

```
N C0X C0Y C1X C1Y T1X T1Y T1R T2X T2Y T2R ...
```

Here, **`N`** is the number of towers, **`(C0X,C0Y)`** is the start of path of the cell phone user, **`(C1X,C1Y)`** is the end of the path, **`(TkX,TkY)`** is the position of the kth tower, and **`TkR`** is its transmission radius. The start and end points of the paths are distinct.

The last problem is terminated by the line

```
0
```

Output

For each configuration, output one line containing the percentage of coverage the cell phone has, rounded to two decimal places.

Sample Input

```
3 0 0 100 0 0 0 10 5 0 10 15 0 10
1 0 0 100 0 40 10 50
0
```

Sample Output

```
25.00
88.99
```

Problem C: XML

Source: `xml.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

In this problem, you are asked to determine if a given document satisfies the syntax of an XML-like language.

A simple XML-like document can be parsed as a sequence of the following:

1. Plain text---ASCII codes between 32 and 127 (inclusive), with none of the following symbols: **`<`**, **`>`**, **`&`**
2. The sequences:
   - **`<`**
   - **`>`**
   - **`&`**

   These encode a **`<`**, **`>`**, or **`&`** respectively.
3. **`&xHEX;`** HEX must be any even (positive) number of upper or lower case hexadecimal digits, and this represents the bytes given.
4. **`<tag>`** Tag can be any lowercase alphanumeric string. This tag is pushed onto the context stack.
5. **`<tag/>`** This tag is not pushed onto the context stack (there is no closing context).
6. **`</tag>`** This tag removes the **`<tag>`** context from the stack, which must be topmost on the stack.

By the time the entire document is parsed, the context stack is empty for a valid document. We should also note that the empty string is considered valid.

Input

You will be given a number of documents to process. Each document is given as one line of text which may be empty. The input is terminated by the end of file.

Output

For each document given, print **`valid`** on a single line if it is a valid XML-like document, or **`invalid`** otherwise.

Sample Input

```
the quick brown fox.
the <i><b>quick</b> brown</i> fox.
<doc>fox &amp; socks.</doc>
3x+5&gt;7
Null: &x00;
<doc>the quick brown fox.
the <i>quick <b>brown</i></b> fox
fox & socks.
3x+5>7
Null: &x0;
```

Sample Output

```
valid
valid
valid
valid
valid
invalid
invalid
invalid
invalid
invalid
```

Problem D: Clickomania

Source: `clickomania.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

Clickomania is a puzzle in which one starts with a rectangular grid of cells of different colours. In each step, a player selects ("clicks") a cell. All connected cells of the same colour as the selected cell (including itself) are removed if the selected cell is connected to at least one other cell of the same colour. The resulting "hole" is filled in by adjacent cells based on some rule, and the object of the game is to remove all cells in the grid.

In this problem, we are interested in the one-dimensional version of the problem. The starting point of the puzzle is a string of colours (each represented by an uppercase letter). At any point, one may select (click) a letter provided that the same letter occurs before or after the one selected. The substring of the same letter containing the selected letter is removed, and the string is shortened to remove the hole created. To solve the puzzle, the player has to remove all letters and obtain the empty string. If the player obtains a non-empty string in which no letter can be selected, then the player loses.

For example, if one starts with the string "ABBAABBAAB", selecting the first "B" gives "AAABBAAB". Next, selecting the last "A" gives "AAABBB". Selecting an "A" followed by a "B" gives the empty string. On the other hand, if one selects the third "B" first, the string "ABBAAAAB" is obtained. One may verify that regardless of the next selections, we obtain either the string "A" or the string "B" in which no letter can be selected. Thus, one must be careful in the sequence of selections chosen in order to solve a puzzle. Furthermore, there are some puzzles that cannot be solved regardless of the choice of selections. For example, "ABBAAAAB" is not a solvable puzzle.

Some facts are known about solvable puzzles:

- The empty string is solvable.
- If **`x`** and **`y`** are solvable puzzles, so are **`xy`**, **`AxA`**, and **`AxAyA`** for any uppercase letter **`A`**.
- All other puzzles not covered by the rules above are unsolvable.

Given a puzzle, your task is to determine whether it can be solved or not.

Input

Each case of input is specified by a single line. Each line contains a string of uppercase letters. Each string has at least one but no more than 150 characters. The input is terminated by the end of file.

Output

For each input case, print **`solvable`** on a single line if there is a sequence of selections that solves the puzzle. Otherwise, print **`unsolvable`** on a line.

Sample Input

```
ABBAABBAAB
ABBAAAAB
```

Sample Output

```
solvable
unsolvable
```

Problem E: Balance

Source: `balance.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

An investor invests a certain percentage of his assets into **`NINSTRUMENTS`** financial instruments. After each term, these instruments deduct a certain fixed administrative cost, followed by a fee that is a percentage of the amount that was invested at the beginning of the term, and then add a return, which is a (positive or negative) percentage of the amount invested at the beginning of the term. If any account drops to zero or below after such a transaction, it is considered closed (no fees are charged against it, and is treated as simply zero) until a rebalancing occurs.

Rebalancing occurs after every **`NREBALANCE`** terms, where the total assets of the investor are redistributed according to the original ratios for the instruments. Without rebalancing, the investor's assets would become dominated by the higher return instruments, which would expose them to more risk compared to a balanced investment plan. Note that it is possible that all instruments drop to zero, in which case they all remain closed for the remaining terms.

You are to model the value of such an investment strategy and report the ending value in each instrument (before rebalancing, if it happens to land on a term when a rebalance is due). Compute your results using double precision (do not round intermediate values to pennies), but round your final answers to pennies.

Input

The first line of the input contains the three positive integers:

```
NINSTRUMENTS NTERMS NREBALANCE
```

There are no more than 10 instruments, and the number of terms is at most 20. This is followed by 3 lines of floating-point numbers separated by spaces, in the following format:

```
FIXED_FEE(1) .. FIXED_FEE(NINSTRUMENTS)
PERCENTAGE_FEE(1) .. PERCENTAGE_FEE(NINSTRUMENTS)
PRINCIPAL_START(1) .. PRINCIPAL_START(NINSTRUMENTS)
```

Finally, there are **`NTERMS`** lines each containing **`NINSTRUMENTS`** floating-point numbers indicating the percentage return of each instrument in each term:

```
RETURN(1,1) .. RETURN(1,NINSTRUMENTS)
RETURN(2,1) .. RETURN(2,NINSTRUMENTS)
.
.
RETURN(NTERMS,1) .. RETURN(NTERMS,NINSTRUMENTS)
```

All percentages (**PERCENTAGE_FEE** and **RETURN**) are given as ratios, up to 4 decimal places. For example, a fee of 0.0002 means 0.02% of the investment in this instrument is deducted as a fee each term. **FIXED_FEE** and **PRINCIPAL_START** are non-negative floating-point numbers that are specified to 2 decimal places. At least one of the **PRINCIPAL_START** values is positive.

Output

Write on a single line the principal of each investment (separated by a space) at the end of **NTERMS** terms. Round each principal to the nearest penny.

```
PRINCIPAL_END(1) .. PRINCIPAL_END(NINSTRUMENTS)
```

Sample Input

```
4 10 5
5.00 10.00 20.00 50.00
0.002 0.001 0.0008 0.0005
150000.00 100000.00 75000.00 50000.00
0.10 0.05 -0.05 -0.85
0.10 0.05 -0.10 -0.85
0.10 0.05 -0.20 -0.85
0.10 0.05 -0.40 -0.85
0.10 0.05 -0.80 -0.85
0.10 0.05 -0.05 -0.90
0.10 0.05 -0.05 -0.90
0.10 0.05 -0.05 -0.90
0.10 0.05 -0.05 -0.85
0.10 0.05 -0.05 -0.85
```

Sample Output

```
237698.69 126086.01 57298.74 0.00
```

Problem F: Resistors

Source: `resistors.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

Every electrical appliance (such as a light bulb) has a certain resistance. If the appliance is connected to a given voltage, the higher its resistance, the lower the current flowing through the appliance. The unit of measurement for resistance is the ohm. In order to avoid round-off errors that can affect floating-point numbers, we will use rational numbers (quotients of positive integers) to represent the resistance of an appliance numerically.

There are two basic ways to connect two or more appliances into a configuration of appliances: serially (Figure 1) or parallel (Figure 2).

Two or more configurations can be further connected serially or parallel to yield another (more complex) configuration yet, and this process of building more complex configurations from existing ones can be repeated any number of times (Figure 3).

In general, a configuration is either a single appliance, or a serial connection of two or more configurations, or a parallel connection of two or more configurations.

The resistance of a configuration of resistors can be computed according to the following two rules:

1. The resistance of a serial configuration is the sum of the resistances of its component configurations.
2. The resistance of a parallel configuration is the reciprocal of the sum of the reciprocals of its component configurations.

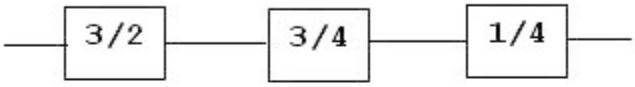

Figure 1

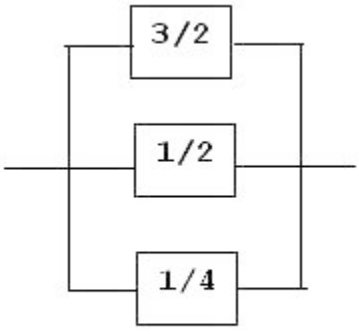

Figure 2

In Figure 1, the resistance of the configuration is 3/2 + 3/4 + 1/4 = 5/2 ohm.

In Figure 2, the resistance of the configuration is 1/(1/(3/2) + 1/(1/2) + 1/ (1/4)) = 3/20 ohm

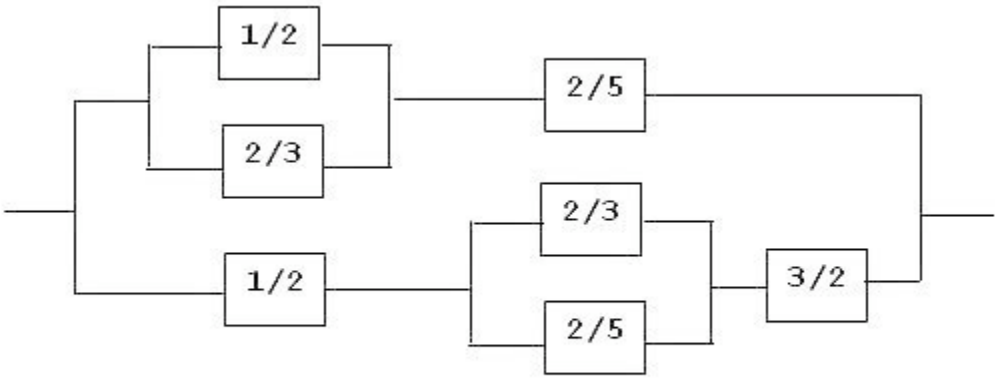

Figure 3

In Figure 3, we first calculate 1/(1/(1/2) + 1/(2/3)) + 2/5 = 24/35 and 1/2 + 1/(1/(2/3) + 1/(2/5)) + 3/2 = 9/4. Adding the reciprocals of these two values and reciprocating the result, we get 72/137 ohm.

A configuration can be represented in text format.

- A single appliance is represented by the numerical value of its resistance (without enclosing parentheses).
- A configuration that is the serial connection of several configurations is represented as a list of the representations of its components, separated by the ampersand character ("&") and enclosed in a pair of parentheses.
- A configuration that is the parallel connection of several configurations is represented as a list of the representations of its components, separated by the vertical bar character ("|") and enclosed in a pair of parentheses.

Thus, figures 1, 2, and 3 are represented in text format by the respective expressions:

```
(3/2 & 3/4 & 1/4)
(3/2 | 1/2 | 1/4)
(((1/2 | 2/3) & 2/5) | (1/2 & (2/3 | 2/5) & 3/2))
```

Input

The input consists of a number of test cases, one test case per line. Each line of the input contains a valid expression that defines a configuration according to the rules stated above. The resistance values of the appliances will be positive rational numbers, in the form **`NUMERATOR/DENOMINATOR`**. There will be one blank space on each side of every ampersand or vertical bar. There will be no other blank spaces in the expression.

Output

For each test case, print the resistance of the configuration on a new line, in the form **`NUMERATOR/DENOMINATOR`**, with all common factors of **`NUMERATOR`** and **`DENOMINATOR`** cancelled. Do not print any blank spaces.

Sample Input

```
15/1
(3/2 & 3/4 & 1/4)
(3/2 | 1/2 | 1/4)
((1/2 | 2/3) & 2/5)
(1/2 & (2/3 | 2/5) & 3/2)
(((1/2 | 2/3) & 2/5) | (1/2 & (2/3 | 2/5) & 3/2))
```

Sample Output

```
15/1
5/2
3/20
24/35
9/4
72/137
```

Problem G: Problem G: Range

Source: `range.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

Some automobiles display the estimated driving range, that is, the distance you can expect to drive it (without adding fuel) before running out of fuel. Here is how it works: periodically, the vehicle's computer records the odometer reading and the weight of fuel in the fuel tank. From this data, the fuel consumption over a certain distance can be computed. From the fuel consumption and the most recent measurement of fuel tank contents (which we assume is current for all practical purposes), the range can be calculated.

Intervals over which the quantity of fuel increased (fuel was added to the tank) will not be used in the computations. For example, in the first problem instance of the sample input, the interval where the fuel weight increased from 29.9 kilograms to 34.2 kilograms will not be used. In this example, 16.3 kilograms of fuel were consumed over a distance of 228.6 kilometers. Therefore, the most recently measured fuel contents of 31.2 kilograms will enable you to drive another 438 kilometers (rounded to the nearest integer). The input will always contain at least one interval (two consecutive lines of input) where no fuel was added to the tank.

Input

The input contains data for a number of problem instances. Each problem instance consists of three or more (odometer reading, fuel weight) pairs, one pair per line. Distances are measured in kilometers and fuel weight in kilograms. All input numbers will be given to one decimal place.

The end of each problem instance will be signaled by a (0.0, 0.0) pair. The last problem instance will be followed by a (-1.0, -1.0) pair.

Output

For each problem instance, print the range, rounded to the nearest integer.

Sample Input

```
18400.5 43.2
18440.4 40.4
18482.7 37.0
18540.2 33.1
18585.3 29.9
18620.8 34.2
18664.6 31.2
0.0 0.0
18400.5 43.2
18440.4 40.4
18482.7 37.0
18540.2 33.1
18585.3 29.9
0.0 0.0
-1.0 -1.0
```

Sample Output

```
438
415
```

Problem H: Papa

Source: `papa.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

While cleaning your parents' attic, you discovered a box containing many documents describing the relationships among your ancestors. Given these documents, you are interested in answering a number of questions about the relationships implied by the document. Fortunately, all your ancestors have unique names so it is possible to make many inferences without any confusion.

It is assumed that all relationships (implied or given) satisfy the following:

- A person can be either male, female, or have an unknown (undetermined by the data set) sex;
- a person can have at most one spouse (of the opposite sex), and X is Y's wife if and only if Y is X's husband;
- a person can have at most one mother and at most one father that can be inferred from the given data;
- if a person has a mother and father, then the mother and father are married;
- the children of a person's spouse are that person's children as well;
- the spouse of a son (or daughter) is not considered to be a daughter (or son). In other words, "sons" and "daughters" refer to biological sons and daughters.

The information provided will be consistent, and you may assume there are no hidden relationships that are not explicitly stated or implied by the above rules of consistency.

Input

The first part of the input consists of a number of lines describing the known relationships. Each relationship is listed on one line in the form:

`NAME is NAME's RELATIONSHIP.`

where **`NAME`** is a lowercase alphabetic word (never 'is'), and **`RELATIONSHIP`** is one of

- **`wife`**
- **`husband`**
- **`daughter`**
- **`son`**
- **`mother`**
- **`father`**

This first part is terminated by a blank line. You may assume that there is at least one known relationship given, and there are at most 100 distinct names mentioned in the known

relationships.

This is followed by a list of questions (one per line) of the form:

is NAME NAME's RELATIONSHIP?

where **NAME** is as before, but **RELATIONSHIP** is one of

- **wife**
- **husband**
- **daughter**
- **son**
- **mother**
- **father**
- **niece**: **X** is a niece of **Y** if there are **W** and **Z** such that **X** is a daughter of **W**, and **W** and **Y** are distinct children of **Z**.
- **nephew**: **X** is a nephew of **Y** if there are **W** and **Z** such that **X** is a son of **W**, and **W** and **Y** are distinct children of **Z**.
- **grandfather**: **X** is a grandfather of **Y** if there is **Z** such that **X** is a father of **Z**, and **Z** is the father or mother of **Y**.
- **grandmother**: **X** is a grandmother of **Y** if there is **Z** such that **X** is a mother of **Z**, and **Z** is the father or mother of **Y**.
- **grandson**: **X** is a grandson of **Y** if there is **Z** such that **X** is a son of **Z**, and **Z** is a child of **Y**.
- **granddaughter**: **X** is a granddaughter of **Y** if there is **Z** such that **X** is a daughter of **Z**, and **Z** is a child of **Y**.

All names appearing in the questions will be mentioned in the list of known relationships. The list of questions is terminated by the end of file.

Output

For each question, print on a line **yes** or **no** if the answer of the question can be determined, or **unknown** if the answer may be yes or no because the sex of the relevant person(s) in the question cannot be determined from the known relationships.

Sample Input

```
john is mary's husband.
john is tom's father.
mary is jane's mother.
jane is anna's mother.

is mary john's wife?
is jane mary's daughter?
is tom mary's husband?
is anna jane's daughter?
```

Sample Output

```
yes
yes
no
unknown
```

Problem I: Wax

Source: `wax.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

You are a flooring contractor with bickering employees. You need to have these employees work together to wax the floor of some rooms, each with 1 door.

None of the workers want to work more than the others, and insist on working together so they can see if someone else is goofing off, so you must assign equal areas of each room to each employee. Each worker is assigned a contiguous portion of the room to wax. The portions assigned are separated by straight line segments that radiate out from the doorway to a wall in the room, so they can exit the room after the job is done.

Input

The input is a series of problem definitions as follows:

WIDTH HEIGHT DOOR WORKERS

Each is a non-negative integer value less than or equal to 100, with at least 2 workers, and a positive **WIDTH** and **HEIGHT**. They specify the positions of the corners of the room: **(0,0)**, **(WIDTH,0)**, **(WIDTH,HEIGHT)**, and **(0,HEIGHT)**, as well as the location of the door: **(DOOR,0)**, **0 < DOOR < WIDTH**. Each problem definition is given on its own line.

The end of input is indicated by a list of 4 zeros:

0 0 0 0

Output

The output are the coordinates, with 3 decimal digits of precision, of the ends of the **WORKERS-1** line segments extending from the door to a wall of the room so that the room is divided into **WORKERS** parts with equal areas. The points should be given in counter-clockwise order.

Sample Input

```
3 5 2 4
10 10 5 2
10 10 5 4
0 0 0 0
```

Sample Output

```
2.500 5.000 1.000 5.000 0.000 3.750
5.000 10.000
10.000 10.000 5.000 10.000 0.000 10.000
```