Quite a problem
Problem ID: quiteaproblem

It gets tiring, looking for all ways in which the word ‘problem’ can be used (and mis-used) in the news media. And yet, that’s been your job for several years: looking through news stories for that word. Wouldn’t it be better if you could automate the process?

Input

Each line of input is one test case. Lines are at most 80 characters long. There are at most 1000 lines of input. Input ends at end of file.

Output

For each line of input, print `yes` if the line contains ‘problem’, and `no` otherwise. Any capitalization of ‘problem’ counts as an occurrence.

Sample Input 1

```
Problematic pair programming
"There’s a joke that pairs, like fish and house guests, go
rotten after three days," said Zach Brock, an engineering
manager.  Working out problems with a pairing partner can be
a lot like working out problems with a significant other.
During one recent rough patch, Jamie Kite, a developer, sat
her partner down for a talk. "Hey, it feels like we’re
driving in different directions," she recalls saying. "It’s
like any relationship," Ms.  Kite said. "If you don’t talk
about the problems, it’s not going to work." When those
timeouts don’t solve the problem, partners can turn to
on-staff coaches who can help with counseling.  "People who
have been pairing a while, they’ll start acting like old
married couples," said Marc Phillips, one of the coaches.
People can be as much of a challenge as writing software.
(Excerpted from "Computer Programmers Learn Tough Lesson in
Sharing"; Wall Street Journal, August 27, 2012)
```

Sample Output 1

```
yes
no
no
yes
yes
no
no
no
no
yes
yes
no
no
no
no
no
no
```

Multi-touch gesture classification
Problem ID: multitouch

With the advent of touch-screen based interfaces, software designers have had to reinvent computer control for finger-based gestures. The most basic task in interpreting a gesture is classifying the touches into one of several possible gestures. This can be difficult because the touches are really blobs of pixels on the screen (not unique points), and gestures involve movement.

We define a *touch* as a connected set of pixels. A pair of pixels is part of the same touch if they are horizontally or vertically adjacent. The *touch point* is the position of a touch. It's defined as the average location of all pixels that are in the set.

A *grip* is the set of all touch points for an image. The *grip point* is the average location of all touch points in the grip. For each touch point we define its *touch vector* as the vector from the grip point to the touch point. A *gesture* is a pair of grips (initial and final), and both grips have the same number of touches. There is a one-to-one correspondence between touches in the initial and final grip. The correspondence is chosen such that the sum of squared distances between corresponding touch points is minimized.

There are three kinds of gestures: pan, zoom, and rotation. There is a distance associated with each kind. The *pan distance* is measured between the two grip points. The *grip spread* is the average distance between each touch point and the grip point. The *zoom distance* is the difference between the two grip spreads. The *touch rotation* is the signed angle (in $(-\pi, \pi)$ radians) between the two touch vectors for corresponding touch points (which is zero if either touch vector has length zero). The *grip rotation* is the average touch rotation. The *rotation distance* is the arc length of the grip rotation along a circle with radius equal the grip spread of the initial grip.

You need to classify the gestures based whether it is a pan, zoom, or rotation. Pick the one which has the greatest associated distance.

Input

Input is one pair of images, given side-by-side. Both images are 15×30 pixels, and there will be a single space on each line separating the two images. Each image uses `X` to represent pixels of a touch, and a period (`.`) to represent no touch. Both images have the same number of touches, between 1 and 5.

Output

Print the number of touches and the type of gesture (pan, zoom, or rotate) as well as the direction of the gesture for zoom or rotate. The direction for a zoom is `in` if the grip spread of the final grip is smaller than that of the initial grip, otherwise it is `out`. The direction of a rotate is either `clockwise` or `counter-clockwise`. All distance comparisons that affect the output will be robust against numerical error of 10^{-5}.

Sample Input 1

```
.............................. ..............................
.............................. ..............................
....XXXX...................... ..............................
....XXXX...................... ............XXXX..............
....XXXX...................... .............XXX..............
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
.............................. ..............................
```

Sample Output 1

```
1 pan
```

Sample Input 2

```
.............................. ..............................
.............................. ..............................
.............................. .................XX...........
.............................. .......XXX......XXXX..........
.............................. ......XXXX........X...........
........XXX....XXX............ ........X.....................
.........XXX...XX............. ..............................
.............................. ..............................
............XX................ ..............................
...........XXX................ ..............................
.............................. ..............................
.............................. ..............XX..............
.............................. .............XXXX.............
.............................. ...............XX.............
.............................. ..............................
```

Sample Output 2

```
3 zoom out
```

Sample Input 3

```
.............................. ..............................
.............................. ..............................
........XX.................... ..............................
........XX.................... .................XXXXX........
.............................. ..................XXX.........
.............................. ..............................
..XXX................XXX...... .....XX.......................
...XX.................XXX..... .....XXX......................
.............................. ..............................
.............................. ..............................
.............................. .....XXX......................
.........XXX.................. ......XX......XXXX............
..........X................... ..............XXXX............
.............................. ..............................
.............................. ..............................
```

Sample Output 3

```
4 rotate counter-clockwise
```

Stacking Curvy Blocks
Problem ID: curvyblocks

You're doing some construction work, and, to save money, you're using some discount, "irregular" construction materials. In particular, you have some blocks that are mostly rectangular, but with one edge that's curvy. As illustrated below, you're going to use these irregular blocks between stacks of ordinary blocks, so they won't shift sideways or rotate. You'll put one irregular block on the bottom, with its curvy edge pointing up, and another above it, with it's curvy edge pointing down. You just need to know how well these blocks fit together. You define the fit quality as the maximum vertical gap between the upper edge of the bottom block and the lower edge of the top block when the upper block is just touching the lower one.

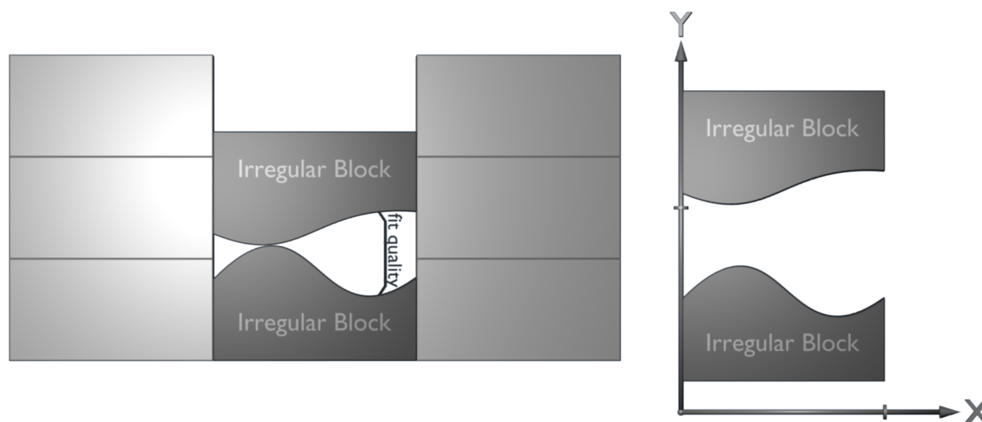


Figure 1: Block stacking and coordinate system

All blocks are one unit wide. You've modeled the curvy edges as cubic polynomials, with the left edge of the block at $x = 0$ and the right edge at $x = 1$.

Input

Each test case is given on two lines, with each line containing four real numbers. The numbers on the first line, b_0 b_1 b_2 b_3, describe the shape of the upper edge of the bottom block. This edge is shaped just like the polynomial $b_0 + b_1x + b_2x^2 + b_3x^3$ for $0 \le x \le 1$. The numbers on the next input line, t_0 t_1 t_2 t_3, describe the bottom edge of the block that's going on top. This edge is shaped just like the polynomial $t_0 + t_1x + t_2x^2 + t_3x^3$ for $0 \le x \le 1$. No input value will have magnitude greater than 50000. There are at most 500 test cases. Input ends at end of file.

Output

For each test case, print out a single line giving the fit quality. An answer is considered correct if its absolute or relative error is at most 10^{-7}.

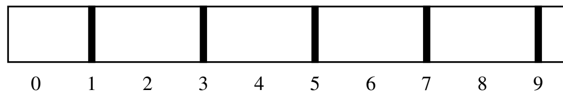
Sample Input 1	Sample Output 1
1.000000 -12.904762 40.476190 -28.571429 3.000000 11.607143 -34.424603 22.817460 2.000000 -10.845238 16.964286 -10.119048 3.000000 4.190476 -3.571429 2.380952	4.396074 6.999999

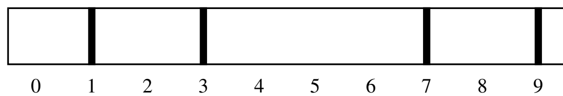
Unicycle counting
Problem ID: unicycles

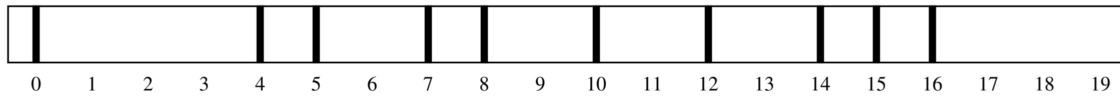
Going to circus college is not as fun as you were led to believe. You are juggling so many classes. Trapeze class is sometimes up, sometimes down. There's a lot of tension in your high-wire class. And you've seen that lion taming can be cat-astrophic.

The one pleasure you find is in riding unicycles with your fellow classmates. Many people have unicycles with different sized wheels. One day you notice that all their tires leave a small mark on the ground, once per rotation. You decide to amuse yourself and avoid your classwork by trying to determine how many unicycles have passed by on a given stretch of road. In fact, you want to know the minimum number of unique unicycles that could have left the marks you observe. You make the simplifying assumption that any unicycle riding on the road will ride completely from the beginning to the end.

The figures below illustrate the sample input. Each thick black vertical stripe represents a mark left by a tire.







Input

Each line of input represents the observations on a stretch of road. A line begins with two integers $1 \leq m \leq 100$ and $1 \leq n \leq 10$, where m represents the length of the road and n represents the number of marks you observe on the road. These are followed by n unique integers $a_1, a_2, \ldots, a_n$, where $0 \leq a_i < m$ for all a_i. These n integers represent the positions where you observed a unicycle's tire has left a mark. There will be at most 100 lines of input. Input ends at end of file.

Output

For each set of observations, print the minimum number of unicycles that could have produced the observed marks.

Sample Input 1	Sample Output 1
10 5 1 3 5 7 9 10 4 1 3 7 9 20 10 0 4 5 7 8 10 12 14 15 16	1 2 3

Choosing Numbers
Problem ID: choosingnumbers

You regularly play a game with friends, and you're tired of losing. The goal of the game is to end up with the largest number in your hand at the end. Initially there is a set of unique numbers on the table. At each turn, a player chooses a number from the table and puts it in his hand. Seems simple, right? However, you may be required to discard numbers in your hand.

During the game, each number can either be on the table, in a player's hand, or in a discard pile. When a player chooses a number x from the table, x is compared with all other numbers y that are not on the table (including other players' hands, your own hand, and those in the discard pile). If x and y have a common factor greater than 1, both are moved to (or remain in) the discard pile. The game ends when all numbers have been chosen from the table.

Input

Each input line describes the set of numbers on the table at the beginning of a game. The line begins with a number $1 \leq n \leq 1000$. Following this are n unique positive integers, all in the range $[2,\ 2 \times 10^9]$. These are the n numbers that are initially on the table. There are at most 1000 games in the input. Input ends at end of file.

Output

For each game print the number x from the table such that choosing x guarantees you win the game. Each game given has a unique winning number.

Sample Input 1	Sample Output 1
2 4 7 4 4 8 15 16 4 2 3 4 8	7 15 3

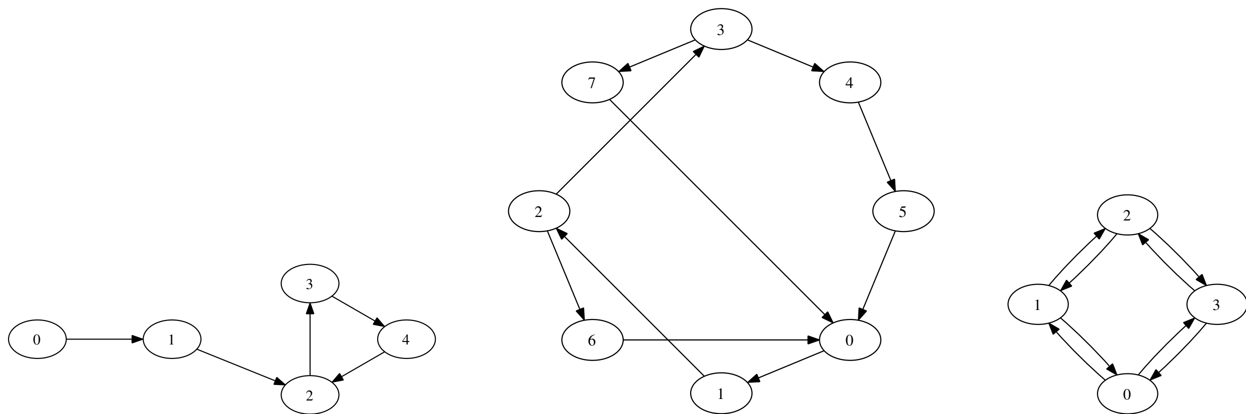
Loopy transit

Problem ID: loopytransit

Luke likes to ride on public transit in different cities he visits, just for fun. He tries to find unique ways to travel in loops: leaving from one transit station, traveling along the transit connections to at least one other station, and returning to the station where he started. He is finding lots of loops, and he wants to know just how many there are in different transit systems. There may be so many he won't ever have time to try them all, but he'll find some satisfaction in knowing they are there.

He's particularly interested in counting *simple loops*. A simple loop is a sequence of unique transit stations $t_1, t_2, \ldots, t_j$, where there's a way to connect directly from t_i to t_{i+1} for $1 \leq i < j$ and also from t_j to t_1. Of course, we can write down a simple loop starting with any of the stations in the loop, therefore we consider any cyclic shift of such a sequence to be the same simple loop. However, two simple loops which visit the same set of transit stations in a different order are considered distinct.

Help Luke by writing a program to count how many unique simple loops there are in each transit system. The following figures illustrate the transit stations (numbered ovals) and one-way connections (arrows) of the sample input.



Input

Input contains a description of one transit system. The description begins with a line containing an integer $3 \leq m \leq 9$ indicating the number of transit stations in the system. Stations are numbered 0 to $m - 1$. The next line contains an integer $1 \leq n \leq m(m-1)$ indicating the number of connections that follow, one connection per line. Each connection is a pair of integers s t ($0 \leq s < m$, $0 \leq t < m$, $s \neq t$), indicating that there is a one-way connection from station s to station t.

Output

Print the number of unique simple loops in the transit system.

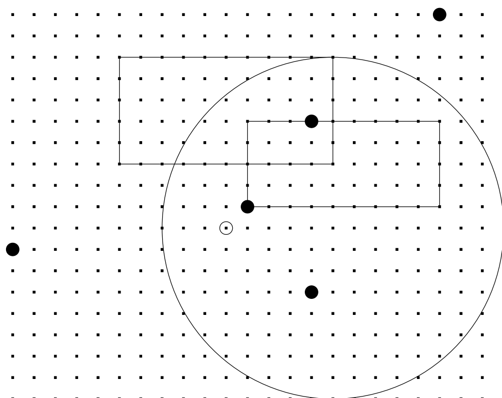
Sample Input 1	Sample Output 1
5 5 0 1 1 2 2 3 3 4 4 2	1

Sample Input 2	Sample Output 2
8 10 0 1 1 2 2 3 3 4 4 5 5 0 2 6 6 0 3 7 7 0	3

Sample Input 3	Sample Output 3
4 8 0 1 1 2 2 3 3 0 1 0 2 1 3 2 0 3	6

Hitting the Targets
Problem ID: hittingtargets

A fundamental operation in computational geometry is determining whether two objects touch. For example, in a game that involves shooting, we want to determine if a player's shot hits a target. A shot is a two dimensional point, and a target is a two dimensional enclosed area. A shot hits a target if it is inside the target. The boundary of a target is inside the target. Since it is possible for targets to overlap, we want to identify how many targets a shot hits.



The figure above illustrates the targets (large unfilled rectangles and circles) and shots (filled circles) of the sample input. The origin $(0, 0)$ is indicated by a small unfilled circle near the center.

Input

Input starts with an integer $1 \leq m \leq 30$ indicating the number of targets. Each of the next m lines begins with the word `rectangle` or `circle` and then a description of the target boundary. A rectangular target's boundary is given as four integers x_1 y_1 x_2 y_2, where $x_1 < x_2$ and $y_1 < y_2$. The points (x_1, y_1) and (x_2, y_2) are the bottom-left and top-right corners of the rectangle, respectively. A circular target's boundary is given as three integers x y r. The center of the circle is at (x, y) and the $0 < r \leq 1000$ is the radius of the circle.

After the target descriptions is an integer $1 \leq n \leq 100$ indicating the number of shots that follow. The next n lines each contain two integers x y, indicating the coordinates of a shot. All x and y coordinates for targets and shots are in the range $[-1000, 1000]$.

Output

For each of the n shots, print the total number of targets the shot hits.

Sample Input 1	Sample Output 1
3 rectangle 1 1 10 5 circle 5 0 8 rectangle -5 3 5 8 5 1 1 4 5 10 10 -10 -1 4 -3	2 3 0 0 1

Out of context
Problem ID: outofcontext

It's that time of year: election season. Political speeches abound, and your friend the armchair pundit likes to find quotes of politicians and use them out of context. You want to help your friend by developing a method to search through text for specified patterns.

One of the more powerful ways to express a text search pattern is using a context-free grammar (CFG). A CFG is used to generate strings, and is defined as a 4-tuple (V, Σ, R, S) where V is a set of variables, Σ is a set of terminal symbols, $S \in V$ is the starting variable, and R is a set of rules. Each rule in R is of the form

$$V \to (V \cup \Sigma)^*$$

which indicates that the head of the rule (the variable to the left of the arrow) can be replaced whenever it appears by the rule's production, a sequence of variables and terminal symbols on the right side of the arrow. It is possible for the right side of a rule to be empty. This indicates that the variable on the left can be replaced by the empty string.

A grammar generates a string of terminals by the process of *derivation*. A derivation begins with a sequence that is just the start variable. Then, until all variables have been removed, we repeatedly replace any variable in the current sequence by any one of that variable's rules.

As an example, here are rules for a grammar with start variable A (left), and an example derivation (right).

$$\begin{aligned} A &\to CFG \\ C &\to CC \\ C &\to \texttt{context} \\ F &\to \texttt{free} \\ F &\to FF \\ G &\to \texttt{grammar} \end{aligned}$$

$$\begin{aligned} A &\Rightarrow CFG \\ &\Rightarrow CCFG \\ &\Rightarrow C\texttt{context}FG \\ &\Rightarrow C\texttt{context}FFG \\ &\Rightarrow C\texttt{context}FF\texttt{grammar} \\ &\Rightarrow C\texttt{contextfree}F\texttt{grammar} \\ &\Rightarrow \texttt{contextcontextfree}F\texttt{grammar} \\ &\Rightarrow \texttt{contextcontextfreefreegrammar} \end{aligned}$$

Write a program that can search through text for substrings that could have been generated by a given CFG.

Input

For this problem, V is the set of English uppercase alphabet letters and Σ is the set of English lowercase alphabet letters. Input begins with a description of R, the rules of the CFG. The first line contains an integer $1 \leq n \leq 30$, indicating the number of rules to follow. The next n lines each describe one rule, in the format

`[A-Z] -> [a-zA-Z]*`

That is, an uppercase letter, a single space, an arrow, a single space, and a string of zero or more uppercase or lowercase letters. Rule productions are at most 10 characters long. The start variable S is the head of the first rule given.

Following the rules are at most 100 lines of text to search, each of length at most 50 characters. Each line of input consists of only English lowercase letters and spaces. Input ends at end of file.

Output

For each line to search, print a line giving the longest non-empty substring that the grammar can generate. If there is a tie for the longest, give the substring that appears earliest on the line. If there is no matching substring, print `NONE` in capital letters.

Sample Input 1

```
5
S -> aSa
S -> bSb
S -> a
S -> b
S ->
where are the abaaba palindromes on this line
none on this line
how about this aaaaaaabbbbbbbbbbbbbbbbba
even a single a or b is a palindrome
```

Sample Output 1

```
abaaba
NONE
abbbbbbbbbbbbbbbbba
a
```

Sample Input 2

```
5
P -> AM
P -> M
A -> NNNx
M -> NNNxNNNN
N -> n
my phone number is nnnxnnnxnnnn what is yours
my number is nnnxnnnn at work
thanks and just in case you can also call
nnnxnnnxnnnn or nnnxnnnxnnnn
```

Sample Output 2

```
nnnxnnnxnnnn
nnnxnnnn
NONE
nnnxnnnxnnnn
```

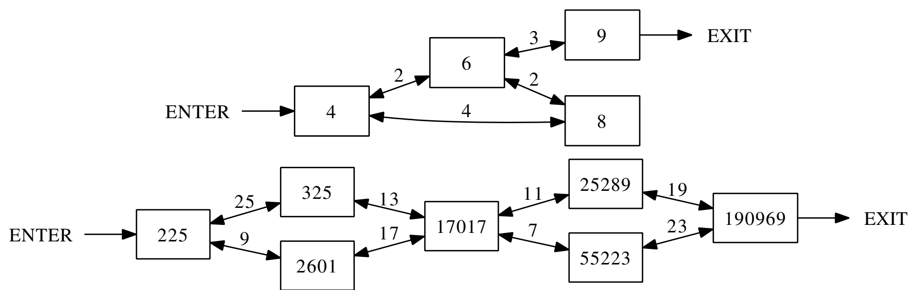
Maze movement
Problem ID: mazemovement

Your boss gave you the task of creating a walking maze, and you are evaluating different designs. Before you commit to one, you want to know how quickly people can move in and out of each different maze. After all, your boss is interested in making money on this venture and, the faster people can move through, the more paying customers you can handle.

A maze is a set of numbered rooms and passages connecting the rooms. The maze's only entrance is at the lowest-numbered room and the only exit is at the highest-numbered room.

Each passage has a limit in the number of people that can pass through at a time. For two rooms numbered x and y, if x and y have a common factor greater than one, then there is a passage between x and y. The largest common factor p is the number of people per minute that can walk from x to y. Simultaneously, p people per minute can also be walking from y to x. The entrance, exit, and rooms can handle any number of people walking through at a time. People want to get through the maze as quickly as possible, so they do not wait around in the rooms.

Here are illustrations of the two sample inputs. Boxes represent the numbered rooms, and each arrow is a passage labeled by the number of people per minute that can walk through it.



Input

Input is a single maze description. The first line is an integer $2 \leq n \leq 1000$ indicating the number of rooms in the maze. This is followed by n unique integers, one per line, which are the room numbers for the maze. Each room number is in the range $[2,\ 2 \times 10^9]$.

Output

Print the maximum number of people per minute that can enter the maze, assuming that people are exiting the maze at the same speed as people entering. No maze supports more than 10^9 people entering per minute.

Sample Input 1	Sample Output 1
4 4 6 8 9	3

Sample Input 2	Sample Output 2
7 25289 17017 2601 325 225 55223 190969	18

Ragged Right

Problem ID: raggedright

Word wrapping is the task of deciding how to break a paragraph of text into lines. For aesthetic reasons, we'd like all the lines except the last one to be about the same length. For example, we would say the text on the left looks less ragged than the text on the right:

`This is a`
`paragraph`
`of text.`
	`This`
	`is a paragraph`
	`of text.`

Your job is to compute a raggedness value for an arbitrary paragraph of text. We'll measure raggedness in a way similar to the TEX typesetting system. Let n be the length, measured in characters, of the longest line of the paragraph. If some other line contains only m characters, then we'll charge a penalty score of $(n - m)^2$ for that line. The raggedness will be the sum of the penalty scores for every line except the last one.

Input

Input consists of a single paragraph of text containing at most 100 lines. Each line of the paragraph contains a sequence of between 1 and 80 characters (letters, punctuation characters, decimal digits and spaces). No line starts or ends with spaces. The paragraph ends at end of file.

Output

Print out a single integer, the raggedness score for paragraph.

Sample Input 1	Sample Output 1

`some blocks`
`of text line up`
`well on the right,`
`but`
`some don't.`
	`283`

Sample Input 2	Sample Output 2

`this line is short`
`this one is a bit longer`
`and this is the longest of all.`
	`218`