1 Saruman's army (`army.{c,cc,java}`)

1.1 Description

Saruman the White must lead his army along a straight path from Isengard to Helm's Deep. To keep track of his forces, Saruman distributes seeing stones, known as *palantirs*, among the troops. Each palantir has a maximum effective range of R units, and must be carried by some troop in the army (i.e., palantirs are not allowed to "free float" in mid-air). Help Saruman take control of Middle Earth by determining the minimum number of palantirs needed for Saruman to ensure that each of his minions is within R units of some palantir.

1.2 Input

The input test file will contain multiple cases. Each test case begins with a single line containing an integer R, the maximum effective range of all palantirs (where $0 \leq R \leq 1000$), and an integer n, the number of troops in Saruman's army (where $1 \leq n \leq 1000$). The next line contains n integers, indicating the positions $x_1, \ldots, x_n$ of each troop (where $0 \leq x_i \leq 1000$). The end-of-file is marked by a test case with $R = n = -1$.

```
0 3
10 20 20
10 7
70 30 1 7 15 20 50
-1 -1
```

1.3 Output

For each test case, print a single integer indicating the minimum number of palantirs needed.

```
2
4
```

In the first test case, Saruman may place a palantir at positions 10 and 20. Here, note that a single palantir with range 0 can cover both of the troops at position 20.

In the second test case, Saruman can place palantirs at position 7 (covering troops at 1, 7, and 15), position 20 (covering positions 20 and 30), position 50, and position 70. Here, note that palantirs must be distributed among troops and are not allowed to "free float." Thus, Saruman cannot place a palantir at position 60 to cover the troops at positions 50 and 70.

2721 - Building Bridges

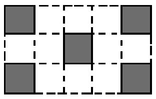
World Finals - Beverly Hills - 2002/2003

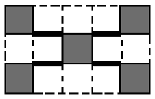
The City Council of New Altonville plans to build a system of bridges connecting all of its downtown buildings together so people can walk from one building to another without going outside. You must write a program to help determine an optimal bridge configuration.

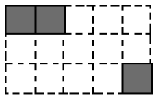
New Altonville is laid out as a grid of squares. Each building occupies a connected set of one or more squares. Two occupied squares whose corners touch are considered to be a single building and do not need a bridge. Bridges may be built only on the grid lines that form the edges of the squares. Each bridge must be built in a straight line and must connect exactly two buildings.

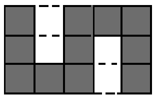
For a given set of buildings, you must find the minimum number of bridges needed to connect all the buildings. If this is impossible, find a solution that minimizes the number of disconnected groups of buildings. Among possible solutions with the same number of bridges, choose the one that minimizes the sum of the lengths of the bridges, measured in multiples of the grid size. Two bridges may cross, but in this case they are considered to be on separate levels and do not provide a connection from one bridge to the other.

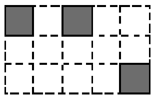
The figure below illustrates four possible city configurations. City 1 consists of five buildings that can be connected by four bridges with a total length of 4. In City 2, no bridges are possible, since no buildings share a common grid line. In City 3, no bridges are needed because there is only one building. In City 4, the best solution uses a single bridge of length 1 to connect two buildings, leaving two disconnected groups (one containing two buildings and one containing a single building).

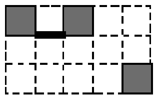

City 1


City 1
with bridges


City 2
No bridges are possible


City 3
No bridges are needed


City 4


City 4
with bridges

Input

The input data set describes several rectangular cities. Each city description begins with a line containing two integers *r* and *c*, representing the size of the city on the north-south and east-west axes measured in grid lengths ($1 \le r \le 100$ and $1 \le c \le 100$). These numbers are followed by exactly *r* lines, each consisting of *c* hash (`#') and dot (`.') characters. Each character corresponds to one square of the grid. A hash character corresponds to a square that is occupied by a building, and a dot character corresponds to a square that is not occupied by a building.

The input data for the last city will be followed by a line containing two zeros.

Output

For each city description, print two or three lines of output as shown below. The first line consists of the city number. If the city has fewer than two buildings, the second line is the sentence `` `No bridges are needed.' ``. If the city has two or more buildings but none of them can be connected by bridges, the second line is the sentence `` `No bridges are possible.' ``. Otherwise, the second line is `` `N bridges of total length L' `` where *N* is the number of bridges and *L* is the sum of the lengths of the bridges of the best solution. (If *N* is 1, use the word `` `bridge' `` rather than `` `bridges.' ``) If the solution leaves two or more disconnected groups of buildings, print a third line containing the number of disconnected groups.

Print a blank line between cases. Use the output format shown in the example.

Sample Input

```
3 5
#...#
..#..
#...#
3 5
##...
.....
....#
3 5
#.###
#.#.#
###.#
3 5
#.#..
.....
....#
0 0
```

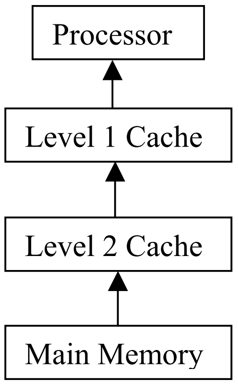
Sample Output

```
City 1
4 bridges of total length 4

City 2
No bridges are possible.
2 disconnected groups

City 3
No bridges are needed.

City 4
1 bridge of total length 1
2 disconnected groups
```

Beverly Hills 2002-2003

Problem 2: Cache Simulation

Most modern computers use one or more levels of cache memory between the processor and the main memory to minimize the time the processor has to wait for information from main memory. Each cache level is characterized as having some number of memory blocks, each of which has a fixed size (measured in bytes, and always a power of 2); the total size of a cache level is just the number of blocks in that cache level times the size of a block. The address of the lowest-numbered byte in each block is always a integral multiple of the block size, and the bytes in a block have contiguous addresses. For example, with a block size of 16, the bytes in a block might possibly be numbered 16 though 31, or 32 through 47, or 160 through 175.



The processor in this problem only reads single bytes, and it does so by issuing a request that specifies the address of the desired byte. If the byte is in the cache closest to the processor (known as the "level 1 cache"), then that cache delivers the byte to the processor; the length of time required for this operation is called the "level 1 access time." If the byte desired by the processor is not in the level 1 cache, but is in the level 2 cache (which is just below the level 1 cache), then the (level 1 size) block containing that byte is delivered from the level 2 cache to the level 1 cache, and then the desired byte is delivered from the level 1 cache to the processor. The total time required in this case is the time required by the level 2 cache to deliver the block to the level 1 cache (naturally called the "level 2 access time"), plus the level 1 access time for the single byte. This pattern continues through all lower cache levels (if present) to the main memory, if necessary. Thus, if a byte requested by the processor isn't in any of the cache levels, the total access time required is the sum of the access times of each cache level plus that of the main memory. The figure to the left illustrates the flow of information in a system with two cache levels.

Each cache is initially empty. When a block is retrieved from a lower-level cache or main memory, it is placed in an empty block in the cache. When no empty blocks are available, and a new block is requested, it will replace an existing block. The particular block it replaces is that block that has been least recently used.

In this problem you will be given the number of caches in a system (between 1 and 3), the block size and total size of each cache, and the access time for each cache and the main memory. Times will be in integral numbers of nanoseconds (nsec). You will then be given a list of the addresses of bytes requested by the processor, and are to compute the time the processor must wait for all of the bytes to be delivered.

As a simple (but unreaslistic) example, suppose the system has two caches. The level 1 cache has 16 byte blocks, a total size of 32 bytes (that is, 2 blocks), and an access time of 4 nanoseconds. The level 2 cache has 32 byte blocks, a total size of 64 bytes (2 blocks), and an access time of 10 nanoseconds. Main memory has an access time of 50 nanoseconds. Suppose the processor requests, in order, bytes from locations 10, 20, 30, 40, and 50. (Cache blocks are numbered here for reference.)

- Since both levels are initially empty, 32 bytes from main memory locations 0 through 31 will be placed in level 2 block 0 (50 nsec), then 16 bytes from that block (addresses 0 through 15) will be placed in level 1 block 0 (10 nsec). Finally, the byte with address 10 will be delivered to the processor (4 nsec). Total time to access the first byte is 64 nsec.
- The byte with address 20 isn't found in level 1, but is found in level 2 in block 0. The 16 bytes containing address 20 (16 to 31) are placed in level 1 block 1 (10 nsec), and the byte with address 20 is delivered to the processor (4 nsec), for a total time of 14 nsec. Note that both blocks in the level 1 cache are now in used.

- Next, the byte with address 30 is sought. Since it is found in the level 1 cache, that byte is simply delivered to the processor (4 nsec).
- Now address 40 is issued by the processor. The byte at this location is not in either cache level, so the corresponding 32 byte block (addresses 32 to 63) is delivered to the level 2 cache and placed there in block 1 (which was previously unused), taking 50 nsec. Then the 16 bytes containing address 40 (32 to 47) are delivered to the level 1 cache (10 nsec more). These 16 bytes are placed in block 0 of the level 1 cache, since it is the least recently used (block 1 of the level 1 cache was used to satisfy the processor's request for address 30). Finally, the byte is delivered to the processor, for a total time of 64 nsec.
- Finally address 50 is requested. Found in the level 2 cache in block 1, the appropriate 16 bytes (48 to 63) are delivered to the level 1 cache (10 nsec). These bytes are placed in block 1 of the level 1 cache, since block 0 was just used. The selected byte is delivered to the processor (4 nsec), for a total time requirement of 14 nsec.

The total time for the bytes at this sequence of addresses to be delivered to the processor is 64 + 14 + 4 + 64 + 14 = 160 nsec.

Input
There will be multiple input cases. For each case, the input begins with an integer that specifies the number of cache levels (between 1 and 3). For each cache level, starting with level 1, the input then contains integers giving the block size, total size, and access time for the cache level. Each cache level has mo more than 100 blocks, and a block size that is no larger than the next (lower level) cache. Next there appears an integer giving the access time for the main memory. Finally, there appears an integer specifying the number of addresses requested by the processor (no more than 1,000) followed by those addresses in the order they were requested; each address is in the range 0 to 65535. A single 0 follows the input for the last case.

Output
For each case, display a single line containing the case number (1, 2, …) and the total time required for all of the bytes requested by the processor to be delivered.

Sample Input

```
2
  16 32 4
  32 64 10
  50
  5 10 20 30 40 50

2
  8 48 4
  32 64 10
  50
  5 10 20 30 40 50

0
```

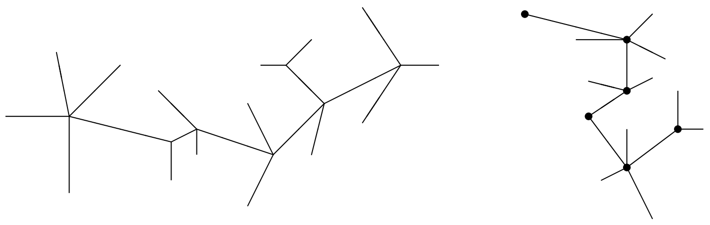
Expected Output

```
Case 1: total time = 160 nanoseconds
Case 2: total time = 170 nanoseconds
```

Problem A: Caterpillar

An undirected graph is called a *caterpillar* if it is connected, has no cycles, and there is a path in the graph where every node is either on this path or a neighbor of a node on the path. This path is called the *spine* of the caterpillar and the spine may not be unique. You are simply going to check graphs to see if they are caterpillars.

For example, the left graph below is not a caterpillar, but the right graph is. One possible spine is shown by dots.



Input

There will be multiple test cases. Each test case starts with a line containing n indicating the number of nodes, numbered 1 through n (a value of $n = 0$ indicates end-of-input). The next line will contain an integer e indicating the number of edges. Starting on the following line will be e pairs n_1 n_2 indicating an undirected edge between nodes n_1 and n_2. This information may span multiple lines. You may assume that $n \leq 100$ and $e \leq 300$. Do not assume that the graphs in the test cases are connected or acyclic.

Output

For each test case generate one line of output. This line should either be

Graph g is a caterpillar.

or

Graph g is not a caterpillar.

as appropriate, where g is the number of the graph, starting at 1.

Sample Input

```
22
21
1 2 2 3 2 4 2 5 2 6 6 7 6 10 10 8 9 10 10 12 11 12 12 13 12 17
18 17 15 17 15 14 16 15 17 20 20 21 20 22 20 19
16
15
1 2 2 3 5 2 4 2 2 6 6 7 6 8 6 9 9 10 10 12 10 11 10 14 10 13 13 16 13 15
0
```

Sample Output

```
Graph 1 is not a caterpillar.
Graph 2 is a caterpillar.
```

7 Censorship (`censorship.{c,cc,java}`)

7.1 Description

As part of the new educational reform program, the CS department has decided to engage in censorship of school texts. In this problem, you must help the department by writing a program which eliminates from an input text string all occurrences of strings from a set of words to be filtered.

More formally, a word w can be removed from another string s if w is a substring of s (i.e., the characters of w appear consecutively in s). Given a text string s and a set T of words to be filtered, return the length of the shortest possible string that can result from iteratively removing words in T from s. Each word in T may be removed from s an unlimited number of times.

7.2 Input

The input test file will contain multiple cases, with each case on a single line of input. Each test case begins with a single integer n (where $1 \leq n \leq 50$) indicating the size of the set T followed by a text string s to be processed. Then, n strings $t_1 \ldots t_n$ indicating the words of T follow. The text string and all of the filter words are guaranteed to contain only the characters 'a' through 'z' and will have lengths between 1 and 50. All filter words will be unique. Input is terminated by a single line containing the number 0; do not process this line. For example:

```
1 ccdedefcde cde
3 aabaab aa ba ab
3 aabaab aa ba bb
0
```

7.3 Output

For each test case, print a single integer indicating the minimum length resulting string possible. For example:

```
1
0
0
```

Possible reductions giving the lengths shown for the three sample inputs are:

$$\begin{aligned}
&\mathrm{c\underline{cde}defcde} \rightarrow \mathrm{\underline{cde}fcde} \rightarrow \mathrm{f\underline{cde}} \rightarrow \mathrm{f}\\
&\mathrm{\underline{aa}baab} \rightarrow \mathrm{\underline{ba}ab} \rightarrow \mathrm{\underline{ab}} \rightarrow \epsilon\\
&\mathrm{\underline{aa}baab} \rightarrow \mathrm{b\underline{aa}b} \rightarrow \mathrm{\underline{bb}} \rightarrow \epsilon,
\end{aligned}$$

where ϵ denotes the empty string.

2 Fibonacci (`fib.{c,cc,java}`)

2.1 Description

In the Fibonacci integer sequence, $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. For example, the first ten terms of the Fibonacci sequence are:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

An alternative formula[1] for the Fibonacci sequence is

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \underbrace{\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdots \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}}_{n \text{ times}}.$$

Given an integer n, your goal is to compute the last 4 digits of F_n.

2.2 Input

The input test file will contain multiple test cases. Each test case consists of a single line containing n (where $0 \leq n \leq$ 1,000,000,000). The end-of-file is denoted by a single line containing the number -1.

```
0
9
999999999
1000000000
-1
```

2.3 Output

For each test case, print the last four digits of F_n. If the last four digits of F_n are all zeros, print '0'; otherwise, omit any leading zeros (i.e., print F_n mod 10000).

```
0
34
626
6875
```

[1]As a reminder, matrix multiplication is associative, and the product of two 2×2 matrices is given by

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}.$$

Also, note that raising any 2×2 matrix to the 0th power gives the identity matrix:

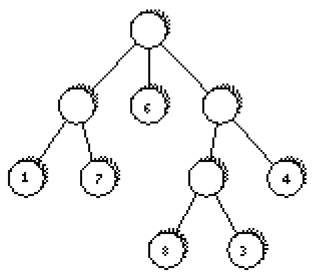
$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Single-Player Games

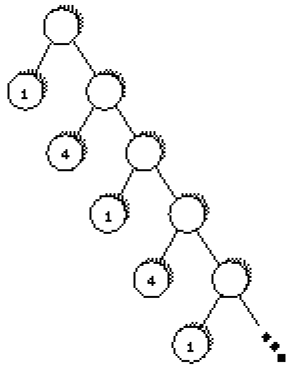
Playing games is the most fun if other people take part. But other players are not always available if you need them, which led to the invention of single-player games. One of the most well-known examples is the infamous ``Solitaire" packaged with Windows, probably responsible for more wasted hours in offices around the world than any other game.

The goal of a single-player game is usually to make ``moves" until one reaches a final state of the game, which results in a win or loss, or a score assigned to that final state. Most players try to optimize the result of the game by employing good strategies. In this problem we are interested in what happens if one plays randomly. After all, these games are mostly used to waste time, and playing randomly achieves this goal as well as any other strategy.

Games can very compactly represented as (possibly infinite) trees. Every node of the tree repre- sents a possible game state. The root of the tree corresponds to the starting position of the game. For an inner node, its children are the game states to which one can move in a single move. The leaf nodes are the final states, and every one of them is assigned a number, which is the score one receives when ending up at that leaf.



```
a = ((1 7) 6 ((8 3) 4))
```



```
a = (1 b)
b = (4 a)
```

Trees are defined using the following grammar.

$$\begin{array}{rcl} Definition & ::= & Identifier\text{“}=\text{”}RealTree \\ RealTree & ::= & \text{“(”}Tree^{+}\text{“)”} \\ Tree & ::= & Identifier \mid Integer \mid \text{“(”}Tree^{+}\text{“)”} \\ Identifier & ::= & \mathtt{a} \mid \mathtt{b} \mid \ldots \mid \mathtt{z} \\ Integer & \in & \{\ldots,-3,-2,-1,0,1,2,3,\ldots,\} \end{array}$$

By using a *Definition*, the *RealTree* on the right-hand side of the equation is assigned to the *Identifier* on the left. A *RealTree* consists of a root node and one or more children, given as a sequence enclosed in brackets. And a *Tree* is either

- the tree represented by a given *Identifier*, or

- a leaf node, represented by a single *Integer*, or
- an inner node, represented by a sequence of one or more *Trees* (its children), enclosed in brackets.

Your goal is to compute the expected score, if one plays randomly, i.e. at each inner node selects one of the children uniformly at random. This expected score is well-defined even for the infinite trees definable in our framework as long as the probability that the game ends (playing randomly) is 1.

Input

The input file contains several gametree descriptions. Each description starts with a line containing the number *n* of identifiers used in the description. The identifiers used will be the first *n* lowercase letters of the alphabet. The following *n* lines contain the definitions of these identifiers (in the order `a`, `b`, ...). Each definition may contain arbitrary whitespace (but of course there will be no spaces within a single integer). The right hand side of a definition will contain only identifiers from the first *n* lowercase letters. The inputs ends with a test case starting with *n* = 0. This test case should not be processed.

Output

For each gametree description in the input, first output the number of the game. Then, for all *n* identifiers in the order `a`, `b`, ..., output the following. If an identifier represents a gametree for which the probability of finishing the game is 1, print the expected score (when playing randomly). This value should be exact to three digits to the right of the decimal point.

If the game described by the variable does not end with probability 1, print ``` ``Expected score for id undefined'' ``` instead. Output a blank line after each test case.

Sample Input

```
1
a = ((1 7) 6 ((8 3) 4))
2
a = (1 b)
b = (4 a)
1
a = (a a a)
0
```

Sample Output

```
Game 1
Expected score for a = 4.917

Game 2
Expected score for a = 2.000
Expected score for b = 3.000

Game 3
Expected score for a undefined
```

Miguel Revilla
2000-05-22

Problem B: Hie with the Pie

The Pizazz Pizzeria prides itself in delivering pizzas to its customers as fast as possible. Unfortunately, due to cutbacks, they can afford to hire only one driver to do the deliveries. He will wait for 1 or more (up to 10) orders to be processed before he starts any deliveries. Needless to say, he would like to take the shortest route in delivering these goodies and returning to the pizzeria, even if it means passing the same location(s) or the pizzeria more than once on the way. He has commissioned you to write a program to help him.

Input

Input will consist of multiple test cases. The first line will contain a single integer n indicating the number of orders to deliver, where $1 \le n \le 10$. After this will be $n+1$ lines each containing $n+1$ integers indicating the times to travel between the pizzeria (numbered 0) and the n locations (numbers 1 to n). The j^{th} value on the i^{th} line indicates the time to go directly from location i to location j without visiting any other locations along the way. Note that there may be quicker ways to go from i to j via other locations, due to different speed limits, traffic lights, etc. Also, the time values may not be symmetric, i.e., the time to go directly from location i to j may not be the same as the time to go directly from location j to i. An input value of $n = 0$ will terminate input.

Output

For each test case, you should output a single number indicating the minimum time to deliver all of the pizzas and return to the pizzeria.

Sample Input

```
3
0 1 10 10
1 0 1 2
10 1 0 10
10 2 10 0
0
```

Sample Output

```
8
```

5 Pool (`pool.{c,cc,java}`)

5.1 Description

Billiards, also commonly known as "pool," is a popular game in North America. The game is played on a rectangular table with six pockets—one at each corner and one in the middle of each of the two longer sides of the table. The object of the game is to strike a cue ball so that it collides with other balls, knocking them into the pockets.

The surface of our pool table measures 108" by 54", and to facilitate computation, we place it on the Cartesian plane with its southwest corner situated at $(0, 0)$, and its northeast corner at $(108, 54)$. Therefore, the centers of the 6 pockets, numbered 1 through 6, will have coordinates of $(0, 0)$, $(54, 0)$, $(108, 0)$, $(0, 54)$, $(54, 54)$, and $(108, 54)$, respectively (see Figure 2). The billiard balls are spherical and measure 2" in diameter.

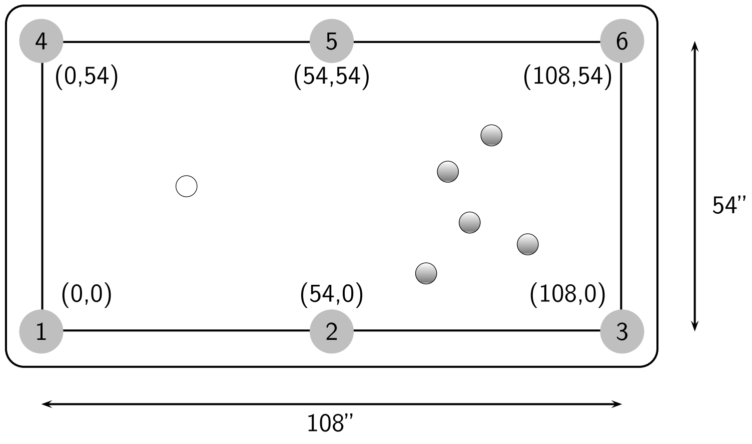


Figure 2: Diagram of pool table.

Given the location of the cue ball, a target ball, and a number of other balls on the table, your task for this problem is to write a program to determine whether or not you can successfully make a particular pool shot. The cue ball can be struck in any direction in a straight line. We consider collisions between the balls to be perfectly elastic, so that the target ball will always travel in a straight line, away from the point on its surface contacted by the cue ball (see Figure 3).[2]

A shot is considered possible if the cue ball can be struck so that it collides directly with the target ball, in turn sending the target ball directly into a pocket. Neither ball should collide with any other balls, bounce off the edges of the table (cushions), nor should their centers cross the boundaries of the table. In other words, you are not to consider any bank shots, combination shots, spin shots, or any other trick shots in this problem. Note that the difference between the incoming angle of the cue ball and the outgoing angle of the target ball must be greater than 90°. The target ball is considered to land in a pocket when its center coincides with the center of that pocket.

[2]You may assume that the cue ball disappears immediately after making contact with the target ball.

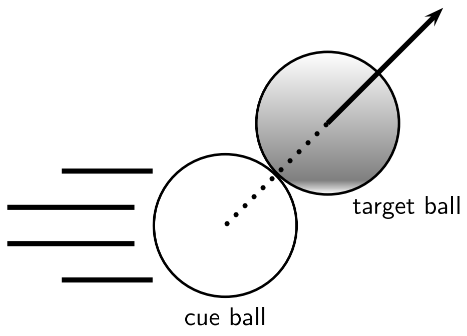


Figure 3: Diagram of pool ball collision.

5.2 Input

The input test file will contain multiple cases. The first line of each test case contains four real numbers, x_c y_c x_t y_t, where (x_c, y_c) is the location of the cue ball and (x_t, y_t) is the location of the target ball. The second line of the test case contains an integer n (where $0 \leq n \leq 14$), the number of additional balls on the table, followed by n pairs of real numbers, x_1 y_1 $\ldots$ x_n y_n, where (x_i, y_i) is the location of the ith additional (possibly obstructing) ball. No two balls overlap, and all balls are strictly in the interior of the table; in particular, all provided coordinates obey $3 < x < 105$ and $3 < y < 51$.

Input is terminated with a single line containing only the number 0; do not process this line. For example:

```
30 27 70 24
0
80 27 40 27
2 38 28 38 26
81.0 27.0 54.5 27.0
1 54.0 33.3
81.0 27.0 54.5 25.0
1 54.0 33.3
0
```

5.3 Output

For each test case, output on a single line the number(s) of the pocket(s) into which the target ball can be shot, sorted in ascending numerical order. Separate pocket numbers with a single space. If there are no pockets for which a clear shot exists, output the words “no shot”. For example:

```
3 6
no shot
1 4
1 2 4
```