Problem A. Elevator Stopping Plan

Input file:	`elevator.in`
Output file:	`elevator.out`

ZSoft Corp. is a software company in GaoKe Hall. And the workers in the hall are very hard-working. But the elevator in that hall always drives them crazy. Why? Because there is only one elevator in GaoKe Hall, while there are hundreds of companies in it. Every morning, people must waste a lot of time waiting for the elevator.

Hal, a smart guy in ZSoft, wants to change this situation. He wants to find a way to make the elevator work more effectively. But it's not an easy job.

There are 31 floors in GaoKe Hall. It takes four seconds for the elevator to raise one floor. It means that it costs $(31 - 1) \times 4 = 120$ seconds if the elevator goes from the first floor to the 31^{st} floor without stop. The elevator also spends 10 second during a single stop. So, if the elevator stops at each floor, it costs $30 \times 4 + 29 \times 10 = 410$ seconds (it is not necessary to calculate the stopping time at 31st floor).

As an alternative, it takes 20 seconds for a worker to go up or down one floor. It takes $30 \times 20 = 600$ seconds for a worker to walk from the 1st floor to the 31st floor. Obviously, it is not a good idea. So some people choose to use the elevator to get a floor which is the nearest to their office.

After thinking over for a long time, Hal finally found a way to improve this situation. He told the elevator man his idea. First, the elevator man asks the people which floors they want to go. He will then design a stopping plan which minimize the time the last person need to arrive the floor where his office locates. For example, if the elevator is required to stop at the 4^{th}, 5^{th} and 10^{th} floors, the stopping plan would require the elevator stops at the 4^{th} and 10^{th} floors. Using this plan, the elevator will arrive at 4^{th} floor in $3 \times 4 = 12$ seconds, then it will stop for 10 seconds, then it will arrive at 10^{th} floor in $3 \times 4 + 10 + 6 \times 4 = 46$ seconds. People who want to go 4^{th} floor will reach their floor in 12 seconds, people who want to go to the 5^{th} floor will reach their floor in $12 + 20 = 32$ seconds and people who want to go to the 10^{th} floor will reach it in 46 seconds. Therefore it takes 46 seconds for the last person to reach his floor. It is a good deal for all people.

Now, you are supposed to write a program to help the elevator man to design the stopping plan, which minimizes the time the last person needs to arrive at his floor.

Input

The input consists of several testcases. Each testcase is in a single line formatted as follows:

$n\ f_1\ f_2\ \ldots\ f_n$

It means that there are totally n floors at which the elevator needs to stop, and $n = 0$ means no testcases any more. $f_1\ f_2\ \ldots\ f_n$ are the floors at which the elevator is to be stopped ($n \le 30$, $2 \le f_1 < f_2 < \ldots < f_n \le 31$). Every number is separated by a single space.

Output

For each testcase, output the time the last person needs in the first line and the stopping floors in the second line. Please note that there should be a total number of the floors in the beginning of the second line. There may be several solutions, any appropriate one is accepted. No extra spaces are allowed.

Sample input and output

`elevator.in`	`elevator.out`

`3 4 5 10`
`1 2`
`0`
	`46`
	`2 4 10`
	`4`
	`1 2`

Problem B. New Go Game

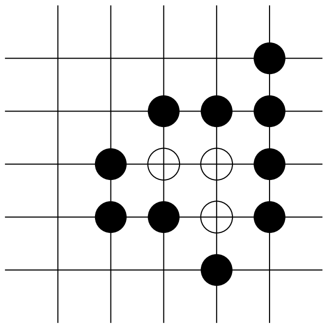
Input file: `go.in`
Output file: `go.out`

The history of Go traces back to some 3 000 years, and the rules have remained essentially unchanged throughout this very long period. The game probably originated in China or the Himalayas. Mythology says that the future of Tibet was once decided over a Go board, when the Buddhist ruler refused to go into battle; instead he challenged the aggressor to a game of Go to avoid bloodshed.

Like Chess, Go is a game of skill, but it differs from Chess in many ways. The rules of Go are very simple, like Chess, it is a challenge to players' analytical skills, but there is far more scope in Go for intuition.

Go is a territorial game. The board, marked with a grid of 19 lines by 19 lines, may be thought of as a piece of land to be shared between the two players. One player has a supply of black pieces, called stones, the other a supply of white. The game starts with an empty board and the players take turns, placing one stone at each turn on a vacant point. Black plays first, and the stones are placed on the intersections of the lines rather than in the squares.

Now, forget the rules of the original Go game. I'll tell you how to play it with only black stones. Let me put some black stones on an empty board first. Then, you are supposed to tell me how many intersections are enclosed by the black stones. What? You can't tell which intersection is enclosed? OK. I'll make it clear. First, an enclosed intersection must be an intersection without any stone on it. Second, an intersection at any border can't be an enclosed intersection. Third, the four near intersections (up, down, left, right) must be enclosed intersections or be occupied by a black stone. On the figure below, there are three enclosed intersections.



It's very easy. But, hold on please. It shouldn't be so easy. For some reason, you don't know where the black stones have been put directly.

Four groups of numbers are used to describe the situation of an $N \times N$ board. In the first group, there are N numbers, and the k-th number indicates the quantity of black stones in the k-th row (from up to down, $1 \le k \le N$). In the second group, there are also N numbers, and the k-th number in this group indicates the quantity of black stones in the k-th column (from left to right, $1 \le k \le N$). In the third group, there are $2N - 1$ numbers, which indicate the quantity of black stones on every slanted line one by one (from left to right, from up to down). In the fourth group, there are also $2N - 1$ numbers, and they indicate the quantity of black stones on every oblique line one by one (from left to right, from down to up). So, the above 5×5 board with several black stones can be described as four groups of numbers (see sample input).

Now, your task is to write a program to rebuild the board from the four groups of numbers (I'm sure you can reconstruct one and only one board from our numbers), and tell me how many enclosed intersections in it.

Input

The input contains several testcases. The first line of each testcase is an integer N ($N \le 15$), representing the size of the board. Then, four lines follow, representing the numbers in the first group, second group and so on.

The input is terminated by a single zero.

Output

For each testcase, output an integer indicating the number of intersections that are enclosed by black stones on the board in a single line. No extra spaces are allowed.

Sample input and output

go.in	go.out

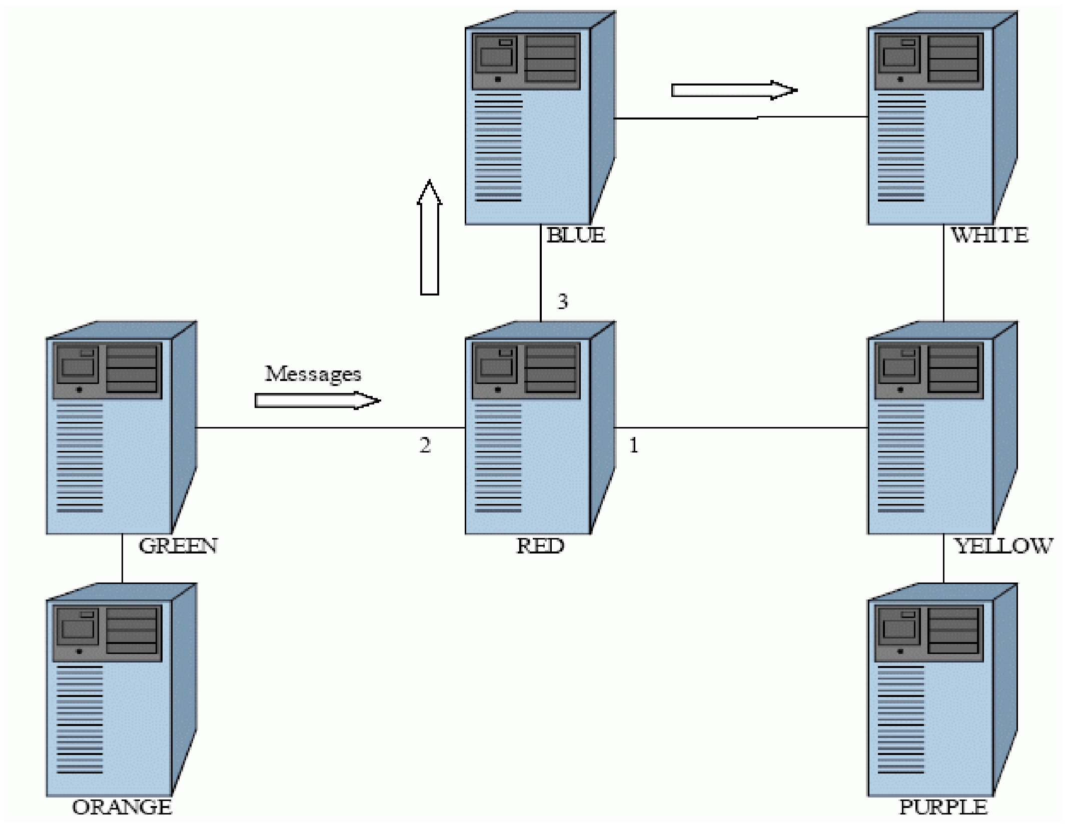
5
1 3 2 3 1
0 2 2 2 4
0 0 1 3 0 2 2 1 1
0 0 0 2 3 2 1 2 0
0
	3

Problem C. Outernet

Input file:	`outernet.in`
Output file:	`outernet.out`

A company named Outdaters is running a small computer wire line network, called Outernet. Not like Internet, Outernet is not based on the TCP/IP protocol. Due to lack of money, not all computers in Outernet can communicate with each other directly.

Outdaters have already found the solution. They created a protocol to make all computers in the network become application proxies. An application proxy can receive data from a connected computer and send them out to another connected computer. Therefore, by using this protocol in Outernet, if a computer wants to send something to a computer not linked directly, it has to send it to a connected computer/application proxy and ask it to help sending them to the destination or another connected computer/application proxy.



The protocol is described as follows.

1. **Port**.

   Application proxies use ports to indicate each connected computer. Port number is an integer number from 0 to 32 767. To an application proxy, 0 means the application proxy itself, each other port number represents a unique computer connected to the application proxy.

2. **Commands**.

   The application proxy accepts only three commands, case sensitive: `TO`, `DATA`, `QUIT`. To each incoming command, application proxy will response with 3-digit result code in a line to the incoming port after handling this command.

   The command format is `xxx`<LF>, where `xxx` is the 3-digit result code and <LF> is line feed.

   There are the following result codes:

- 100: OK. No error/Data routed to destination.
- 101: OK. Data routed to application. (Destination computer is application proxy itself).
- 200: Session ends (Response to `QUIT` command).
- 300: Unknown command.
- 301: Unknown destination.
- 302: No session began.
- 303: Looping not allowed (when the incoming port is the same as the outgoing port).

Details for each command:

(a) `TO`:<destination computer name><LF>

Tells the application proxy that data (from the subsequent commands) need to be sent to <destination computer name>, and cancels the last `TO` command's effect (sends a `QUIT` command to the original destination computer). If `TO` command fails (result code is neither 100 nor 101), the state of the application proxy will not be changed.

Possible result codes are:

- 100: The destination computer is found in routing table, and not the application proxy itself.
- 101: The destination computer is found in routing table, and IS the application proxy itself.
- 301: The destination computer is not found in routing table.
- 303: The destination computer is found in routing table, but the incoming port is the same as the outgoing port.

(b) `DATA`<LF><the data><a dot "."><LF>

Send <the data> to the destination computer. <the data> should be regarded as a data stream, and should be sent to the destination without any alteration if the destination computer is not the application proxy itself. The backslash ("\") is the meta character, "\." means a simple dot ".", instead of then end indicator, and "\\" means "\".

Possible result codes are:

- 100: The destination computer is found in routing table, and not the application proxy itself. The data is routed to the corresponding outgoing port.
- 101: The destination computer is found in routing table, and IS the application proxy itself. The data is routed to the application running on this application proxy.
- 302: No session began, this command is ignored.

(c) `QUIT`<LF>

End this communication session. Possible result codes are:

- 200: Session ends.
- 302: No session began, this command is ignored.

3. **Session**.

When a computer (the requester) sends a `TO` command to an application proxy, a communication session begins; when a `QUIT` command is sent to the application proxy, the session ends. In a session, the requester can send multiple `TO` and `DATA` commands to an application proxy to send out multiple messages. An application proxy is able to handle sessions simultaneously from different ports.

4. **Routing table**.

Each application proxy holds a routing table. It uses this table to find which port should be used for the destination computer name. Each line in the routing table contains two fields, the first is the destination computer name, and the second is the outgoing port number. It means that the data to be sent to a computer with the destination computer name, will be sent out via the port with the outgoing port number. Port number 0 means, the data should be routed to the application running on this application proxy; that destination computer name is actually the application proxy's name.

5. **Routing**.

   Application proxies use the same `TO`, `DATA`, `QUIT` commands to route the incoming data if the routing is possible.

   After searching on the routing table, if the outgoing port is found, application proxies must create a complete session on the outgoing port for each valid incoming `TO` command: one `TO` command at the beginning, zero or more `DATA` commands to route the data, one `QUIT` command in the end if the incoming session ends or another incoming `TO` command is received.

   Port 0 is handled as same as other outgoing ports except that no actually outgoing command is sent, i.e. all the commands' result code will be sent to the incoming port, but no commands will be sent to any outgoing port.

Now, Outdaters hires you to write the engine to implement the protocol for the application proxy.

Input

The input consists of a sequence of testcases. Each begins with a routing table of an application proxy and then the incoming requests of the application proxy.

A routing table includes, in order, a line with an integer M ($1 \leq M \leq 32\,768$), the number of lines in the routing table, and M lines, each of which is a routing line. Each routing line contains a unique destination computer name (1 to 15 alphanumeric characters in the routing table), and then the outgoing port number (an integer from 0 to 32 767), separated by a space, and the computer names are case sensitive.

The incoming requests of the application proxy include several request sessions from the connected computers. A line starts with a number sign "#" and then an integer P (-1 or 1 to 32 767), means the following input is from port P, $P < 0$ means the testcase finishes. The commands in request sessions will not be broken by the "#" lines. To simplify the input handling, data commands in our input file will just contain characters "0"–"9", "a"–"z", "A"–"Z", "@", "#", "_", "+", "–", "*", "/" , "\", "?", ",", "." and <LF>.

The input is terminated by a single zero

Output

For each testcase, print all the outputs of the ports sending out data, in the order of the corresponding input. For each port's output, a line starts with a number sign "#" and then an integer P (-1, 1 to 32 767), means the following commands are output in port P, $P = -1$ means the output of the current testcase finishes. Following the "#" line is the command's output in this port till another "#" line. A "#" line is needed only when the port number need to be changed.

Sample input and output

`outernet.in`	`outernet.out`
5	#2
RED 0	TO:GREEN
YELLOW 1	#1
GREEN 2	100
BLUE 3	#2
WHITE 3	DATA
#1	HELLO
TO:GREEN	.
DATA	#1
HELLO	100
.	#3
#4	TO:WHITE
TO:WHITE	#4
#1	100
Quit	#1
QUIT	300
#2	#2
TO:GREEN	QUIT
DATA	#1
A JOKE to myself	200
.	#2
QUIT	303
#3	302
TO:ORANGE	302
QUIT	#3
#4	301
QUIT	302
#-1	QUIT
0	#4
	200
	#-1

Problem D. Amphiphilic Carbon Molecules

Input file:	`amphiphilic.in`
Output file:	`amphiphilic.out`

Shanghai Hypercomputers, the world's largest computer chip manufacturer, has invented a new class of nanoparticles called Amphiphilic Carbon Molecules (ACMs). ACMs are semiconductors. It means that they can be either conductors or insulators of electrons, and thus possess a property that is very important for the computer chip industry. They are also amphiphilic molecules, which means parts of them are hydrophilic while other parts of them are hydrophobic. Hydrophilic ACMs are soluble in polar solvents (for example, water) but are insoluble in non-polar solvents (for example, acetone). Hydrophobic ACMs, on the contrary, are soluble in acetone but insoluble in water. Semiconductor ACMs dissolved in either water or acetone can be used in the computer chip manufacturing process.

As a materials engineer at Shanghai Hypercomputers, your job is to prepare ACM solutions from ACM particles. You go to your factory everyday at 8 A.M. and find a batch of ACM particles on your workbench. You prepare the ACM solutions by dripping some water, as well as some acetone, into those particles and watch the ACMs dissolve in the solvents. You always want to prepare unmixed solutions, so you first separate the ACM particles by placing an Insulating Carbon Partition Card (ICPC) perpendicular to your workbench. The ICPC is long enough to completely separate the particles. You then drip water on one side of the ICPC and acetone on the other side. The ICPC helps you obtain hydrophilic ACMs dissolved in water on one side and hydrophobic ACMs dissolved in acetone on the other side. If you happen to put the ICPC on top of some ACM particles, those ACMs will be right at the border between the water solution and the acetone solution, and they will be dissolved. Fig. 1 shows your working situation.

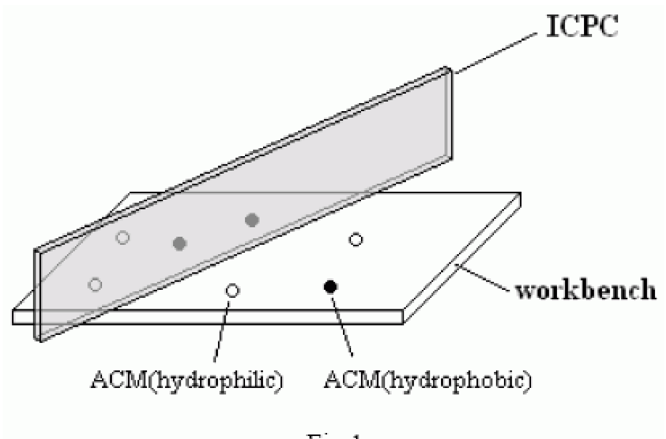


Fig.1

Your daily job is very easy and boring, so your supervisor makes it a little bit more challenging by asking you to dissolve as much ACMs into solution as possible. You know you have to be very careful about where to put the ICPC since hydrophilic ACMs on the acetone side, or hydrophobic ACMs on the water side, will not dissolve. As an experienced engineer, you also know that sometimes it can be very difficult to find the best position for the ICPC, so you decide to write a program to help you. You have asked your supervisor to buy a special digital camera and have it installed above your workbench, so that your program can obtain the exact positions and species (hydrophilic or hydrophobic) of each ACM particle in a 2D pictures taken by the camera. The ICPC you put on your workbench will appear as a line in the 2D pictures.

Input

There will be no more than 10 test cases. Each case starts with a line containing an integer N, which is the number of ACM particles in the test case. N lines then follow. Each line contains three integers x, y, r, where (x, y) is the position of the ACM particle in the 2D picture and r can be 0 or 1, standing for the hydrophilic or hydrophobic type ACM respectively. The absolute value of x, y will be no larger than 10 000. You may assume that N is no more than 1000. $N = 0$ signifies the end of the input and need not be processed. Fig. 2 shows the positions of ACM particles and the best ICPC position for the last test case in the sample input.

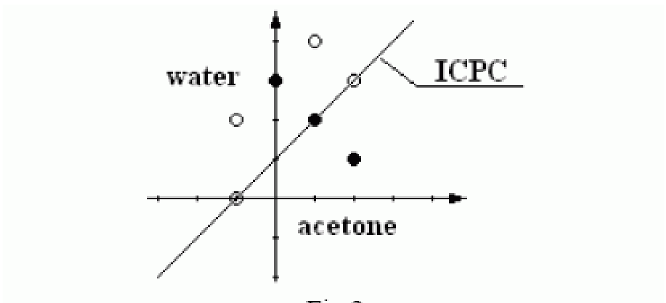


Fig.2

Output

For each test case, output a line containing a single integer, which is the maximum number of dissolved ACM particles.

Sample input and output

amphiphilic.in	amphiphilic.out

3
0 0 0
0 1 0
2 2 1
4
0 0 0
0 4 0
4 0 0
1 2 1
7
-1 0 0
1 2 1
2 3 0
2 1 1
0 3 1
1 4 0
-1 2 0
0
	3
	3
	6

Problem E. Different Digits

Input file:	`digits.in`
Output file:	`digits.out`

Given a positive integer n, your task is to find a positive integer m, which is a multiple of n, and that m contains the least number of different digits when represented in decimal. For example, number 1334 contains three different digits 1, 3 and 4.

Input

The input consists of no more than 50 test cases. Each test case has only one line, which contains a positive integer n ($1 \le n < 65536$). There are no blank lines between cases. A line with a single 0 terminates the input.

Output

For each test case, you should output one line, which contains m. If there are several possible results, you should output the smallest one. Do not output blank lines between cases.

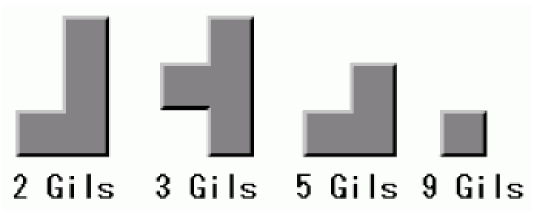
Sample input and output

`digits.in`	`digits.out`
7 15 16 101 0	7 555 16 1111

Problem F. The Floor Bricks

Input file:	`floor.in`
Output file:	`floor.out`

Robert decided to decorate his new room with a cool pattern on the floor, composed with colorful floor bricks. After several days' work, he finally felt satisfied with the pattern he created with the odd shaped bricks. Soon, he found a problem. As the border of the pattern is not a perfect rectangle, the floor is not filled with bricks completely. However, since the shapes of the bricks are quite odd, it is not an easy work to fill the floor with these bricks completely. (Although a brick with a unit size is provided, this kind of brick is quite expensive usually. Fig. 1 shows an example of a set of bricks) As Robert was very proud of his brick pattern, he refuses to modify even one brick in his pattern in order to satisfy the need of filling the floor completely. Instead, he asked you to find solutions for him to fill the rest part of the floor completely with the given bricks so that the Gils needed to buy these bricks are minimized. Of course, the bricks can not overlap each other to fill the floor. Please note that the bricks can be rotated, but cannot be flipped over. In addition, you may assume that all kinds of bricks can be contained within a 3×3 square box.



As the pattern of bricks covers most areas of the floor, the uncovered part of the floor is actually located at the bottom border of the rectangular shaped floor. Therefore, an uncovered part like the one in Fig. 2 can be described with a series of integers, which represents the number of square blocks missed in each column, starting from the left. Thus, the shape in Fig. 2 can be described with 11 integers: 2 2 1 2 3 5 2 3 3 4 1. In addition, you may assume that these integers are no larger than 5. Fig. 3 shows a solution to fill the floor with the brick set in Fig.1 that costs minimal Gils.

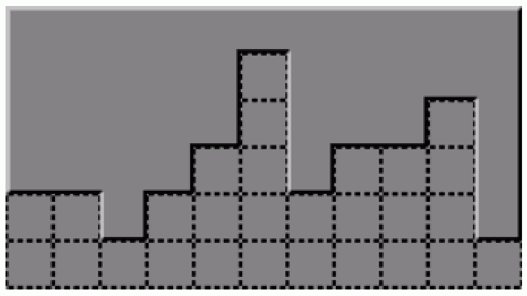


Fig.2

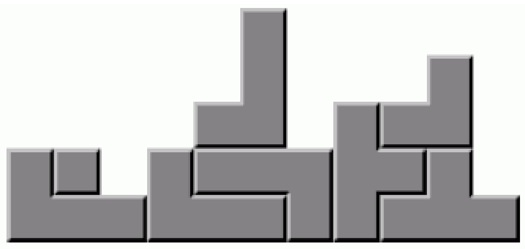


Fig.3

Input

The input consists of at most 20 test cases. The first line of each test case contains an integer n ($1 \le n \le 1000$), which is the width of the floor. The second line contains n integers, separated by single spaces. These integers describe the shape of the uncovered floor as mentioned above. The third line contains an integer m ($1 \le m \le 100$), which is the number of bricks available. After this is the detailed description of these m bricks.

Each brick description consists of 4 lines. The first line is a positive integer, which is the price of that brick in Gils. After this line, there are three lines, each consisting of three characters, which describe the shape of the brick. A dot character represents a space in the shape and a sharp character "#" represents a square block in the shape. You can be sure that the blocks are always connected to form the brick.

There is a line containing a zero after the last test case, which signifies the end of the input and should not be processed.

Output

For each test case, output a line “Need at least g Gil(s).”, where g is the minimal Gils needed to fill the floor with the given bricks. If the floor cannot be filled with the given bricks, output “Impossible.” instead. Do not output blank lines between cases.

Sample input and output

floor.in

```
11
1 4 3 3 2 5 3 2 1 2 2
4
2
#..
#..
##.
3
.#.
.##
.#.
5
...
#..
##.
9
...
.#.
...
1
1
1
1
..#
...
...
1
1
1
1
.##
...
...
0
```

floor.out

```
Need at least 28 Gil(s).
Need at least 1 Gil(s).
Impossible.
```

Problem G. The Rotation Game

Input file:	`game.in`
Output file:	`game.out`

The rotation game uses a # shaped board, which can hold 24 pieces of square blocks (see Fig. 1). The blocks are marked with symbols 1, 2 and 3, with exactly 8 pieces of each kind.

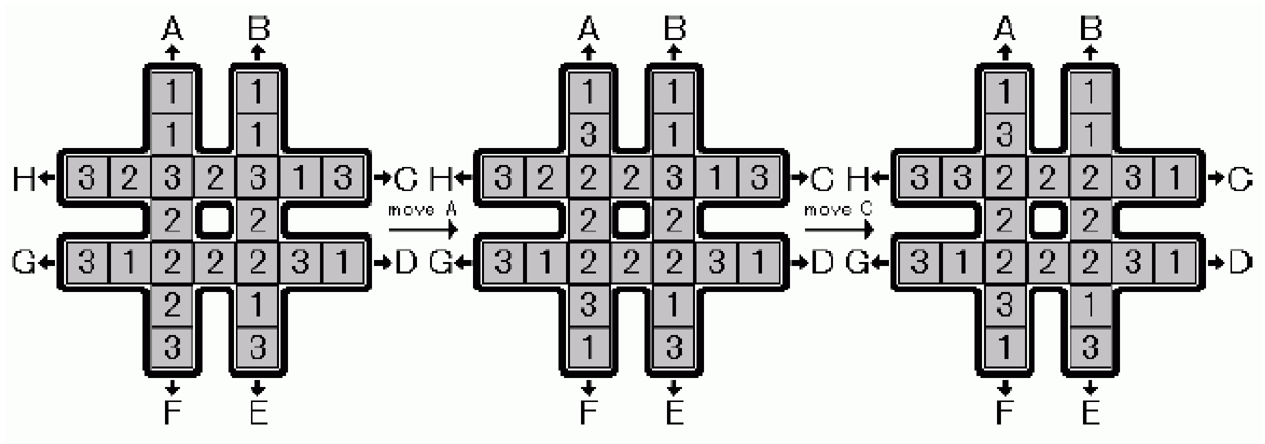


Fig.1

Initially, the blocks are placed on the board randomly. Your task is to move the blocks so that the eight blocks placed in the center square have the same symbol marked. There is only one type of valid move, which is to rotate one of the four lines, each consisting of seven blocks. That is, six blocks in the line are moved towards the head by one block and the head block is moved to the end of the line. The eight possible moves are marked with capital letters A to H. Fig. 1 illustrates two consecutive moves, move A and move C from some initial configuration.

Input

The input consists of no more than 30 test cases. Each test case has only one line that contains 24 numbers, which are the symbols of the blocks in the initial configuration. The rows of blocks are listed from top to bottom. For each row the blocks are listed from left to right. The numbers are separated by spaces. For example, the first test case in the sample input corresponds to the initial configuration in Fig. 1. There are no blank lines between cases. There is a line containing a single 0 after the last test case that ends the input.

Output

For each test case, you must output two lines. The first line contains all the moves needed to reach the final configuration. Each move is a letter, ranging from A to H, and there should not be any spaces between the letters in the line. If no moves are needed, output “No moves needed” instead. In the second line, you must output the symbol of the blocks in the center square after these moves. If there are several possible solutions, you must output the one that uses the least number of moves. If there is still more than one possible solution, you must output the solution that is smallest in dictionary order for the letters of the moves. There is no need to output blank lines between cases.

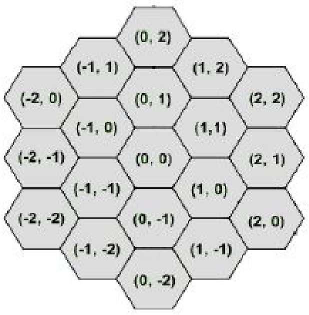
Sample input and output

game.in	game.out
`1 1 1 1 3 2 3 2 3 1 3 2 2 3 1 2 2 2 3 1`	`AC`
`2 1 3 3`	`2`
`1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3 3 3`	`DDHH`
`3 3 3 3`	`2`
`0`	

Problem H. Hexagonal Sticks

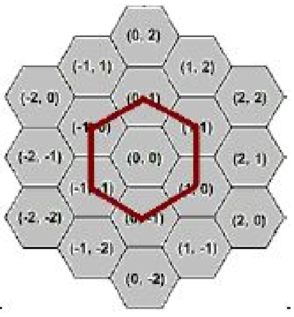
Input file:	`sticks.in`
Output file:	`sticks.out`

In this problem, we will consider an infinite hexagonal grid. The grid consists of equal regular hexagonal cells arranged in the fashion shown below. The figure also elucidates the coordinate system used to identify each cell. Each cell of the grid can be empty or blocked.

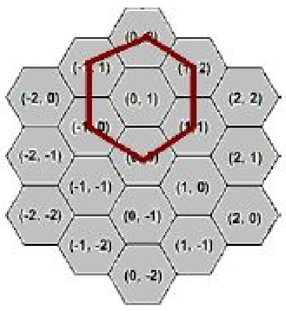


There are few sticks placed randomly on this grid. The length of each stick is one "hexagonal unit". This means, the end points of a stick lies on the centers of neighboring cells. Your job is to move the sticks so that a closed regular hexagonal figure is formed.

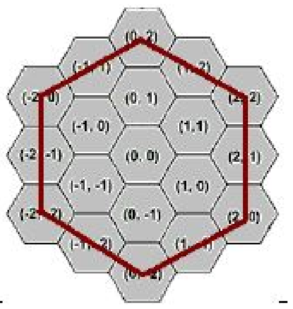
The diagrams below show some closed hexagonal figures made of sticks.



Hexagon with 6 sticks



Hexagon with 6 sticks

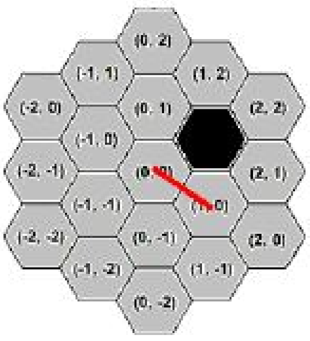


Hexagon with 12 sticks

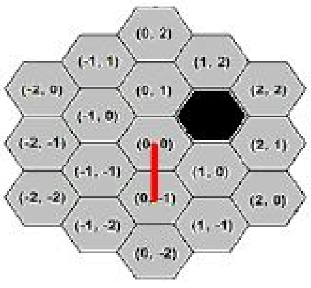
You will be given the initial coordinates of the sticks that are placed on the grid. You will also be given the coordinates of the cells that are blocked. In each move you can do one of the followings:

- Select one stick and throw it out.
- Select one stick and rotate it 60 degrees clockwise/anti-clockwise about one of the end points.
- Select one stick and push it along the length of the stick.

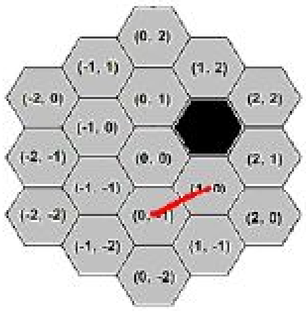
The sticks can never occupy a cell that is blocked. However, two sticks can occupy the same cells at the same time.



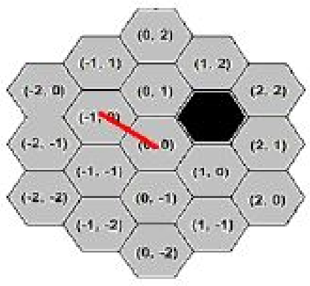
Consider the scenario shown above. We have an obstacle situated at coordinate $(1, 1)$ and a stick at coordinate $(0, 0) \to (1, 0)$. The four possible moves that can be made are depicted below:



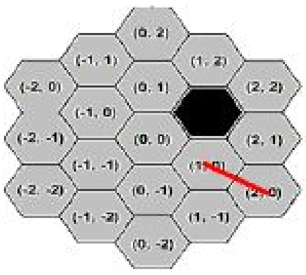
Rotate 60 degrees cw about $(0, 0)$



Rotate 60 degrees a-cw about $(1, 0)$



Push along the length



Push along the length

After all the moves are made, you have to ensure a closed regular hexagonal shape is formed with no other sticks lying around. This means the grid should contain exactly $6 \times x$ sticks where x is positive integer. Can you do this in minimum number of moves?

Input

The first line of input is an integer T ($T < 50$) that indicates the number of test cases. Each case starts with a non-negative integer S ($S < 9$) that gives you the number of sticks available. The next S lines give you the coordinates of the sticks. The coordinates of the sticks will be of the format $x_1\ y_1\ x_2\ y_2$, which means there is a stick from (x_1, y_1) to (x_2, y_2). The coordinates will be valid and the length of each stick will be one hexagonal unit as mentioned above.

The next line will give you a non-negative integer B ($B < 20$) that indicates the number of obstacles. Each of the next B lines will give you the coordinates of the obstacles. The coordinate will be of the format $x_1\ y_1$. It is guaranteed that the given blocks will not overlap with any given sticks. All the coordinates (sticks and blocks) will have values in the range $[-4, 4]$.

Note: Remember that we are dealing with infinite grids. So in the optimal result, it could be possible that the hexagonal sticks lie outside $[-4, 4]$.

Output

For each case, output the case number first followed by the minimum number of moves required. If it is impossible to form a hexagonal grid, output "impossible" instead. Adhere to the sample input/output for exact format.

Sample input and output

sticks.in

```
3
6
-1 -1 -1 0
-1 0 0 1
0 1 1 1
1 1 1 0
1 0 0 -1
0 -1 -1 -1
0
5
-1 0 0 1
0 1 1 1
1 1 1 0
1 0 0 -1
0 -1 -1 -1
0
7
-2 -2 -2 -1
-1 -1 -1 0
0 0 1 1
0 0 1 1
0 0 1 1
1 1 2 2
1 2 2 2
2
1 0
2 0
```

sticks.out

```
Case 1: 0
Case 2: impossible
Case 3: 9
```

Problem I. Hex Tile Equations

Input file: `hex.in`
Output file: `hex.out`

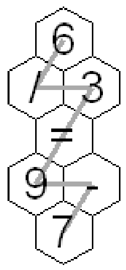


Figure 1

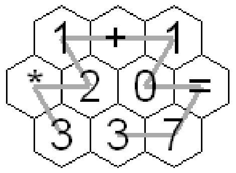


Figure 2

An amusing puzzle consists of a collection of hexagonal tiles packed together with each tile showing a digit or "=" or an arithmetic operation "+", "-", "*", or "/". Consider continuous paths going through each tile exactly once, with each successive tile being an immediate neighbor of the previous tile. The object is to choose such a path so the sequence of characters on the tiles makes an *acceptable* equation, according to the restrictions listed below. A sequence is illustrated in each figure above. In Figure 1, if you follow the gray path from the top, the character sequence is "6/3=9-7". Similarly, in Figure 2, start from the bottom left 3 to get "3*21+10=73".

There are a lot of potential paths through a moderate sized hex tile pattern. A puzzle player may get frustrated and want to see the answer. Your task is to automate the solution.

The arrangement of hex tiles and choices of characters in each puzzle satisfy these rules:

1. The hex pattern has an odd number of rows greater than 2. The odd numbered rows will all contain the same number of tiles. Even numbered rows will have one more hex tile than the odd numbered rows and these longer even numbered rows will stick out both to the left and the right of the odd numbered rows.
2. There is exactly one "=" in the hex pattern.
3. There are no more than two "*" characters in the hex pattern.
4. There will be fewer than 14 total tiles in the hex pattern.
5. With the restrictions on allowed character sequences described below, there will be a unique acceptable solution in the hex pattern.

To have an acceptable solution from the characters in some path, the expressions on each side of the equal sign must be in acceptable form and evaluate to the same numeric value. The following rules define acceptable form of the expressions on each side of the equal sign and the method of expression evaluation:

6. The operators "+", "-", "*", and "/" are only considered as binary operators, so no character sequences where "+" or "-" would be a unary operator are acceptable. For example "-2*3=-6" and "1=5+-4" are not acceptable.
7. The usual precedence of operations is not used. Instead all operations have equal precedence and operations are carried out from left to right. For example "44-4/2=2+3*4" is acceptable and "14=2+3*4" is not acceptable.

8. If a division operation is included, the equation can only be acceptable if the division operation works out to an exact integer result. For example "10/5=12/6" and "7+3/5=3*4/6" are acceptable. "5/2*4=10" is not acceptable because the sides would only be equal with exact mathematical calculation including an intermediate fractional result. "5/2*4=8" is not acceptable because the sides of the equation would only be equal if division were done with truncation.

9. At most two digits together are acceptable. For example, "123+1=124" is not acceptable.

10. A character sequences with a "0" directly followed by another digit is not acceptable. For example, "3*05=15" is not acceptable.

With the assumptions above, an acceptable expression will never involve an intermediate or final arithmetic result with magnitude over three million.

Input

The input will consist of one to fifteen data sets, followed by a line containing only 0.

The first line of a dataset contains blank separated integers r c, where r is the number of rows in the hex pattern and c is the number of entries in the odd numbered rows. The next r lines contain the characters on the hex tiles, one row per line. All hex tile characters for a row are blank separated. The lines for odd numbered rows also start with a blank, to better simulate the way the hexagons fit together. Properties 1–5 apply.

Output

There is one line of output for each data set. It is the unique acceptable equation according to rules 6–10 above. The line includes no spaces.

Sample input and output

hex.in	hex.out

```
5 1
 6
/ 3
 =
9 -
 7
3 3
 1 + 1
* 2 0 =
 3 3 7
5 2
 9 -
* 2 =
 3 4
+ 8 3
 4 /
0
```
	```
	6/3=9-7
	3*21+10=73
	8/4+3*9-2=43
	```

Problem J. Line & Circle Maze

Input file:	`maze.in`
Output file:	`maze.out`

A deranged algorithms professor has devised a terrible final exam: he throws his students into a strange maze formed entirely of linear and circular paths, with line segment endpoints and object intersections forming the junctions of the maze. The professor gives his students a map of the maze and a fixed amount of time to find the exit before he floods the maze with xerobiton particles, causing anyone still in the maze to be immediately inverted at the quantum level. Students who escape pass the course; those who don't are trapped forever in a parallel universe where the grass is blue and the sky is green.

The entrance and the exit are always at a junction as defined above. Knowing that clever ACM programming students will always follow the shortest possible path between two junctions, he chooses the entrance and exit junctions so that the distance that they have to travel is as far as possible. That is, he examines all pairs of junctions that have a path between them, and selects a pair of junctions whose shortest path distance is the longest possible for the maze (which he rebuilds every semester, of course, as the motivation to cheat on this exam is very high).

The joy he derives from quantumly inverting the majority of his students is marred by the tedium of computing the length of the longest of the shortest paths (he needs this to know to decide how much time to put on the clock), so he wants you to write a program to do it for him. He already has a program that generates the mazes, essentially just a random collection of line segments and circles. Your job is to take that collection of line segments and circles, determine the shortest paths between all the distinct pairs of junctions, and report the length of the longest one.

The input to your program is the output of the program that generates his mazes. That program was written by another student, much like yourself, and it meets a few of the professor's specifications:

1. No endpoint of a line segment will lie on a circle.
2. No line segment will intersect a circle at a tangent.
3. If two circles intersect, they intersect at exactly two distinct points.
4. Every maze contains at least two junctions; that is, a minimum maze is either a single line segment, or two circles that intersect.

There is, however, one bug in the program. (He would like to have it fixed, but unfortunately the student who wrote the code never gave him the source, and is now forever trapped in a parallel universe). That bug is that the maze is not always entirely connected. There might be line segments or circles, or both, off by themselves that intersect nothing, or even little "submazes" composed of intersecting line segments and circles that as a whole are not connected to the rest of the maze. The professor insists that your solution account for this! The length that you report must be for a path between connected junctions!

Input

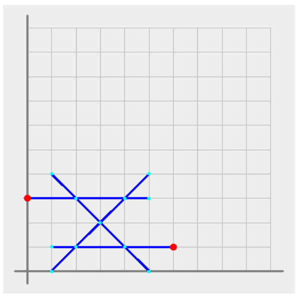
An input test case is a collection of line segments and circles. A line segment is specified as "L X_1 Y_1 X_2 Y_2" where "L" is a literal character, and (X_1, Y_1) and (X_2, Y_2) are the line segment endpoints. A circle is specified by "C X Y R" where "C" is a literal character, (X, Y) is the center of the circle, and R is its radius. All input values are integers, and line segment and circle objects are entirely contained in the first quadrant within the box defined by $(0, 0)$ at the lower left and $(100, 100)$ at the upper right. Each test case will consist of from 1 to 20 objects, terminated by a line containing only a single asterisk. Following the final test case, a line containing only a single asterisk marks the end of the input.

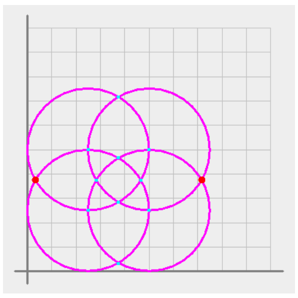
Output

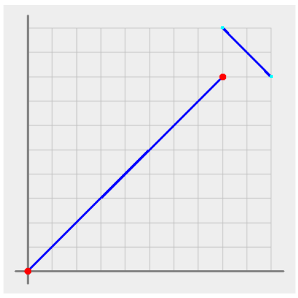
For each input maze, output "Case N: ", where N is the input case number starting at one (1), followed by the length, rounded to one decimal, of the longest possible shortest path between a pair of connected junctions.

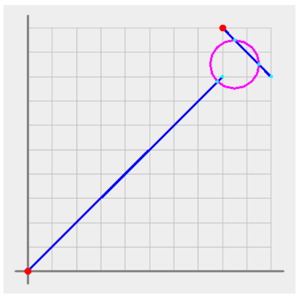
Sample input and output

maze.in	maze.out
`L 10 0 50 40`	`Case 1: 68.3`
`L 10 40 50 0`	`Case 2: 78.5`
`L 10 10 60 10`	`Case 3: 113.1`
`L 0 30 50 30`	`Case 4: 140.8`
`*`	
`C 25 25 25`	
`C 50 25 25`	
`C 25 50 25`	
`C 50 50 25`	
`*`	
`L 0 0 80 80`	
`L 80 100 100 80`	
`*`	
`L 0 0 80 80`	
`L 80 100 100 80`	
`C 85 85 10`	
`*`	
`*`	


Test case 1


Test case 2


Test case 3


Test case 4

Problem K. Enigmatic Travel

Input file:	`enigmatic.in`
Output file:	`enigmatic.out`

Suhan and Laina live in an n-dimensional city where there are $n+1$ locations. Any two locations (consider these locations as points) are equidistant from each other and connected by only one bi-directional road. They love to roam together around the city on their favourite bi-verbal (a kind of vehicle). Kiri, a tenth generation robot also lives in the same city and wants to kill Suhan out of jealousy. That is why Suhan and Laina are very careful about keeping their thoughts and plans secret. Therefore nobody knows:

- where Suhan and Laina lives;
- what their destination location is;
- which roads they will use.

So their journey can start from any location, ends in another location and they may use any road sequence they like. Their destination location may be same or different than the source location. For example when their tour is guaranteed to be a simple cycle their source and destination location are same.

Given the number of locations in the city (L) you will have to find the expected cost (often considered as average) of one of their single travelling. You can assume that the cost of travelling from one location to another through the direct (also shortest) path is one universal joule.

Input

The input file contains several lines of input. Each line contains a single integer L ($15 \geq L > 2$) that indicates the number of locations in the city. Input is terminated by a line where value of L is zero. This line should not be processed.

Output

For each line of input produce one line of output. This line contains three floating-point numbers F_1, F_2, F_3. Here F_1 is the expected cost when they travel along a path, F_2 is the expected cost when it is guaranteed that they travel along a simple path and F_3 is the expected cost when it is guaranteed that they travel along a simple cycle. All the floating point numbers should be rounded up to four digits after the decimal point. You should assume that their travelling cost never exceeds L for any possible travel. Travelling cost is always expressed in universal joules.

Sample input and output

enigmatic.in	enigmatic.out

3
4
5
0
	2.4286 1.5000 3.0000
	3.5500 2.2000 3.5000
	4.6716 3.0625 4.2000