Problem A
A Careful Approach
Input: approach.in

If you think participating in a programming contest is stressful, imagine being an air traffic controller. With human lives at stake, an air traffic controller has to focus on tasks while working under constantly changing conditions as well as dealing with unforeseen events.

Consider the task of scheduling the airplanes that are landing at an airport. Incoming airplanes report their positions, directions, and speeds, and then the controller has to devise a landing schedule that brings all airplanes safely to the ground. Generally, the more time there is between successive landings, the "safer" a landing schedule is. This extra time gives pilots the opportunity to react to changing weather and other surprises.

Luckily, part of this scheduling task can be automated – this is where you come in. You will be given scenarios of airplane landings. Each airplane has a time window during which it can safely land. You must compute an order for landing all airplanes that respects these time windows. Furthermore, the airplane landings should be stretched out as much as possible so that the minimum time gap between successive landings is as large as possible. For example, if three airplanes land at 10:00am, 10:05am, and 10:15am, then the smallest gap is five minutes, which occurs between the first two airplanes. Not all gaps have to be the same, but the smallest gap should be as large as possible.

Input
The input file contains several test cases consisting of descriptions of landing scenarios. Each test case starts with a line containing a single integer n ($2 \le n \le 8$), which is the number of airplanes in the scenario. This is followed by n lines, each containing two integers a_i, b_i, which give the beginning and end of the closed interval $[a_i, b_i]$ during which the i^{th} plane can land safely. The numbers a_i and b_i are specified in minutes and satisfy $0 \le a_i \le b_i \le 1440$.

The input is terminated with a line containing the single integer zero.

Output
For each test case in the input, print its case number (starting with 1) followed by the minimum achievable time gap between successive landings. Print the time split into minutes and seconds, rounded to the closest second. Follow the format of the sample output.

Sample Input	Output for the Sample Input
3 0 10 5 15 10 15 2 0 10 10 20 0	Case 1: 7:30 Case 2: 20:00

Problem B
My Bad
Input file: bad.in

A logic circuit maps its input through various gates to its output with no feedback loops in the circuit. The input and output are an ordered set of logical values, represented here by ones and zeros. The circuits we consider are comprised of *and* gates (which output 1 only when their two inputs are both 1), *or* gates (which output 1 when one or both of their inputs are 1), *exclusive or* (xor) gates (which output 1 only when exactly one of the two inputs is 1), and *not* gates (which output the complement of their single input). The figures below show two circuits.

Figure 1: Circuit with 2 gates

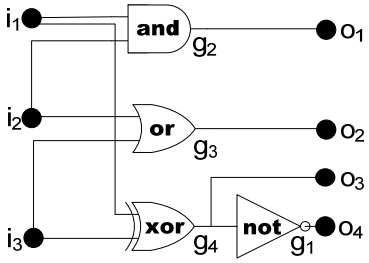

Figure 2: Circuit with 4 gates

Unfortunately, real gates sometimes fail. Although the failures may occur in many different ways, this problem limits attention to gates that fail in one of three ways: 1) always inverting the correct output, 2) always yielding 0, and 3) always yielding 1. In the circuits for this problem, at most one gate will fail.

You must write a program that analyzes a circuit and a number of observations of its input and output to see if the circuit is performing correctly or incorrectly. If at least one set of inputs produces the wrong output, your program must also attempt to determine the unique failing gate and the way in which this gate is failing. This may not always be possible.

Input

The input consists of multiple test cases, each representing a circuit with input and output descriptions. Each test case has the following parts in order.

1. A line containing three positive integers giving the number of inputs ($N \leq 8$), the number of gates ($G \leq 19$), and the number of outputs ($U \leq 19$) in the circuit.
2. One line of input for each gate. The first line describes gate g_1. If there are several gates, the next line describes gate g_2, and so on. Each of these lines contains the gate type (`a` = *and*, `n` = *not*, `o` = *or*, and `x` = *exclusive or*), and identification of the input(s) to the gate. Gate input comes from the circuit inputs (`i1`, `i2`, …) or the output of another gate (`g1`, `g2`, …).
3. A line containing the numbers of the gates connected to the U outputs u_1, u_2, …. For example, if there are three outputs, and u_1 comes from g_5, u_2 from g_1, and u_3 from g_4, then the line would contain: `5 1 4`
4. A line containing an integer which is the number of observations of the circuit's behavior (B).
5. Finally B lines, each containing N values (ones and zeros) giving the observed input values and U values giving the corresponding observed output values. No two observations have the same input values.

Consecutive entries on any line of the input are separated by a single space. The input is terminated with a line containing three zeros.

Output

For each circuit in the input, print its case number (starting with 1), followed by a colon and a blank, and then the circuit analysis, which will be one of the following (with # replaced by the appropriate gate number):

```
No faults detected
Gate # is failing; output inverted
Gate # is failing; output stuck at 0
Gate # is failing; output stuck at 1
Unable to totally classify the failure
```

The circuits pictured in Figure 1 and Figure 2 are used in the first and last sample test cases.

Sample Input	**Output for the Sample Input**
2 2 1 o i1 i2 n g1 2 2 1 0 0 0 0 1 2 1 1 a i1 i2 1 1 1 0 1 2 1 1 a i1 i2 1 2 1 0 1 1 1 1 1 1 1 n i1 1 2 1 1 0 0 3 4 4 n g4 a i1 i2 o i2 i3 x i3 i1 2 3 4 1 4 0 1 0 0 1 0 1 0 1 1 0 1 1 0 1 1 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0	Case 1: No faults detected Case 2: Unable to totally classify the failure Case 3: Gate 1 is failing; output stuck at 1 Case 4: Gate 1 is failing; output inverted Case 5: Gate 2 is failing; output stuck at 0

Problem C
The Return of Carl
Input file: carl.in

Carl the ant is back! When we last left him (Problem A, 2004 World Finals), Carl was a little mixed-up, always taking strange, zigzag paths when traveling. But now, Carl has straightened out his life – literally. He now always takes the straightest, shortest path between any pair of points. This sounds simple, except for one small thing: Carl now spends most of his time on a paperweight in the shape of a regular octahedron. You may recall that an octahedron is one of the five Platonic solids and consists of eight equilateral triangles, as shown in Figure 3.

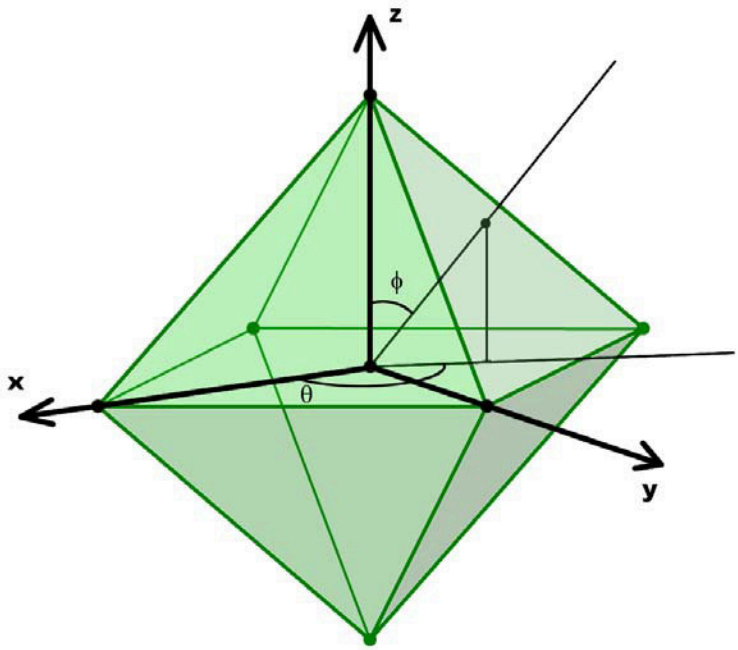

Figure 3: Regular octahedron

Carl has an innate (some say in-ant) ability to always take the shortest path when going from any starting point to any destination point on the paperweight. Your job is to verify this by determining the length of the shortest path when given two such points, not necessarily distinct.

Input

The input contains multiple test cases. Each test case consists of four integers θ_1, ϕ_1, θ_2 and ϕ_2, where $0 \le \theta_i < 360$ and $0 \le \phi_i \le 180$. The first two are the spherical coordinates of the start point, and the last two are the spherical coordinates of the destination point. As shown in Figure 3, θ_i is the *azimuth* and ϕ_i is the *zenith angle*, both of them given in degrees.

The input is terminated by a line containing four negative ones.

The paperweight is fixed for all test cases as follows: the octahedron is centered on the origin and each vertex lies on one of the axes. Every edge is exactly 10 cm long. You should suppose that Carl's size is zero and ignore any supporting mechanisms that may be necessary to hold the paperweight in the correct position.

Output

For each test case, print the case number (starting with 1) and the length in centimeters of the shortest path from the start point to the destination point, rounded to the nearest thousandth. Follow the format of the sample output.

Sample Input	Output for the Sample Input
0 90 90 90 0 90 90 45 0 0 0 180 -1 -1 -1 -1	Case 1: 10.000 Case 2: 8.660 Case 3: 17.321

Problem D
Conduit Packing
Input File: conduit.in

Allied Conduit Manufacturing (ACM) makes metal conduit tubes with round cross-sections that enclose many different types of wires. The circular cross-section of a wire can have a diameter up to 20 millimeters (20000 micrometers). ACM needs a program to compute the minimum diameter of a conduit that can hold 4 wires with specified diameters.

Figure 4 shows examples of fitting four wires of different sizes into conduits of minimum diameters.

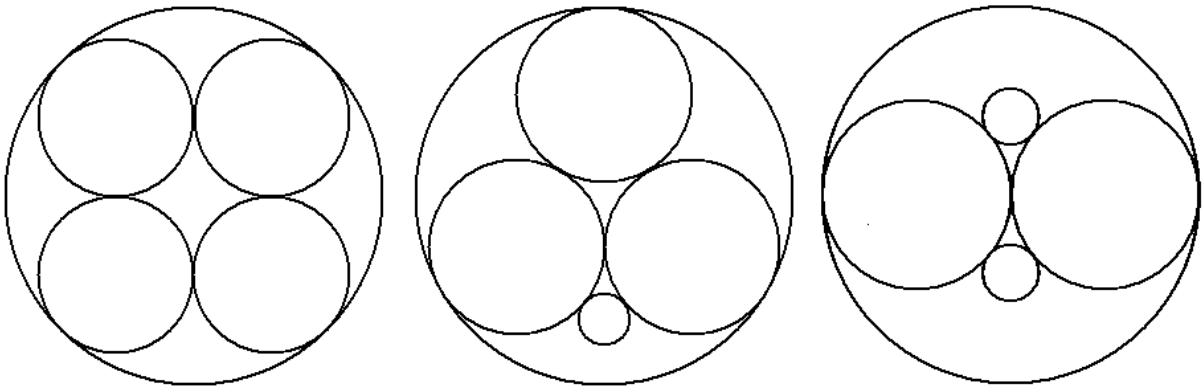

Figure 4: Fitting wires inside conduits

Your program must take the diameters of wires and determine the minimum inside diameter of the conduit that can hold the wires.

Input

The input file contains several test cases. Each test case consists of a line with four integers, d_1, d_2, d_3, and d_4, which are the diameters of the wires in micrometers. The integers satisfy $20000 \geq d_1 \geq d_2 \geq d_3 \geq d_4 > 0$. The last test case is followed by a line containing a single integer zero.

Output

For each test case, print the number of the test case (starting with 1) followed by the minimum conduit diameter in micrometers, rounded to the nearest integer. Follow the format of the sample output.

Sample Input	Output for the Sample Input
10000 10000 10000 10000 10000 10000 10000 3000 12000 12000 3600 3600 0	Case 1: 24142 Case 2: 21547 Case 3: 24000

Problem E
Fare and Balanced
Input: fare.in

Handling traffic congestion is a difficult challenge for young urban planners. Millions of drivers, each with different goals and each making independent choices, combine to form a complex system with sometimes predictable, sometimes chaotic behavior. As a devoted civil servant, you have been tasked with optimizing rush-hour traffic over collections of roads.

All the roads lie between a residential area and a downtown business district. In the morning, each person living in the residential area drives a route to the business district. The morning commuter traffic on any particular road travels in only one direction, and no route has cycles (morning drivers do not backtrack).

Each road takes a certain time to drive, so some routes are faster than others. Drivers are much more likely to choose the faster routes, leading to congestion on those roads. In order to balance the traffic as much as possible, you are to add tolls to some roads so that the perceived "cost" of every route ends up the same. However, to avoid annoying drivers too much, you must not levy a toll on any driver twice, no matter which route he or she takes.

Figure 5 shows a collection of five roads that form routes from the residential area (at intersection 1) to the downtown business district (at intersection 4). The driving cost of each road is written in large blue font. The dotted arrows show the three possible routes from 1 to 4. Initially the costs of the routes are 10, 8 and 12. After adding a toll of cost 2 to the road connecting 1 and 4 and a toll of cost 4 to the road connecting 3 and 4, the cost of each route becomes 12.

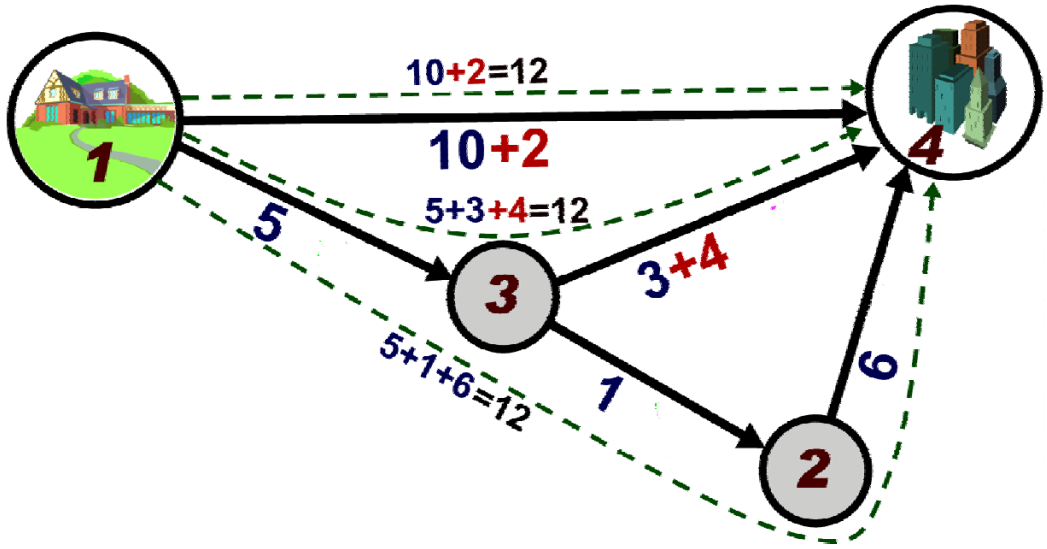

Figure 5: Roads connecting residential area at intersection 1 to business district at intersection 4

You must determine which roads should have tolls and how much each toll should be so that every route from start to finish has the same cost (driving time cost + possible toll) and no route contains more than one toll road. Additionally, the tolls should be chosen so as to minimize the final cost. In some settings, it might be impossible to impose tolls that satisfy the above conditions.

Input

Input consists of several test cases. A test case starts with a line containing an integer N ($2 \le N \le 50000$), which is the number of road intersections, and R ($1 \le R \le 50000$), which is the number of roads. Each of the next R lines contains three integers x_i, y_i, and c_i ($1 \le x_i, y_i \le N$, $1 \le c_i \le 1000$), indicating that morning traffic takes road i from intersection x_i to intersection y_i with a base driving time cost of c_i. Intersection 1 is the starting residential

area, and intersection N is the goal business district. Roads are numbered from 1 to R in the given input order. Every intersection is part of a route from 1 to N, and there are no cycles.

The last test case is followed by a line containing two zeros.

Output

For each test case, print one line containing the case number (starting with 1), the number of roads to toll (T), and the final cost of every route. On the next T lines, print the road number i and the positive cost of the toll to apply to that road. If there are multiple minimal cost solutions, any will do. If there are none, print `No solution`. Follow the format of the sample output.

Sample Input	Output for Sample Input
4 5 1 3 5 3 2 1 2 4 6 1 4 10 3 4 3 3 4 1 2 1 1 2 2 2 3 1 2 3 2 0 0	Case 1: 2 12 4 2 5 4 Case 2: No solution

Problem F
Deer-Proof Fence
Input: fence.in

Uncle Magnus has planted some young saplings on his farm as part of his reforestation project. Unfortunately, deer like to eat tender sapling shoots and leaves, making it necessary to build protective fences around them. Since deer and other sapling nibblers can reach partway over the fence, every fence must lie at least a minimum distance (a *margin*) from each sapling.

Deer-proof fencing is quite expensive, so Uncle Magnus wants to minimize the total length of fencing used. Your job is to write a program that computes the minimum length of fencing that is required to enclose and protect the saplings. Fences may include both straight and curved segments. You may design a single fence that encloses all saplings or multiple fences that enclose separate groups of saplings.

Figure 6 shows two example configurations, each consisting of three saplings with different margin requirements. In the top configuration, which corresponds to the first sample input, the minimum-length solution consists of two separate fences. In the bottom configuration, which corresponds to the second sample input, the minimum-length solution consists of a single fence.

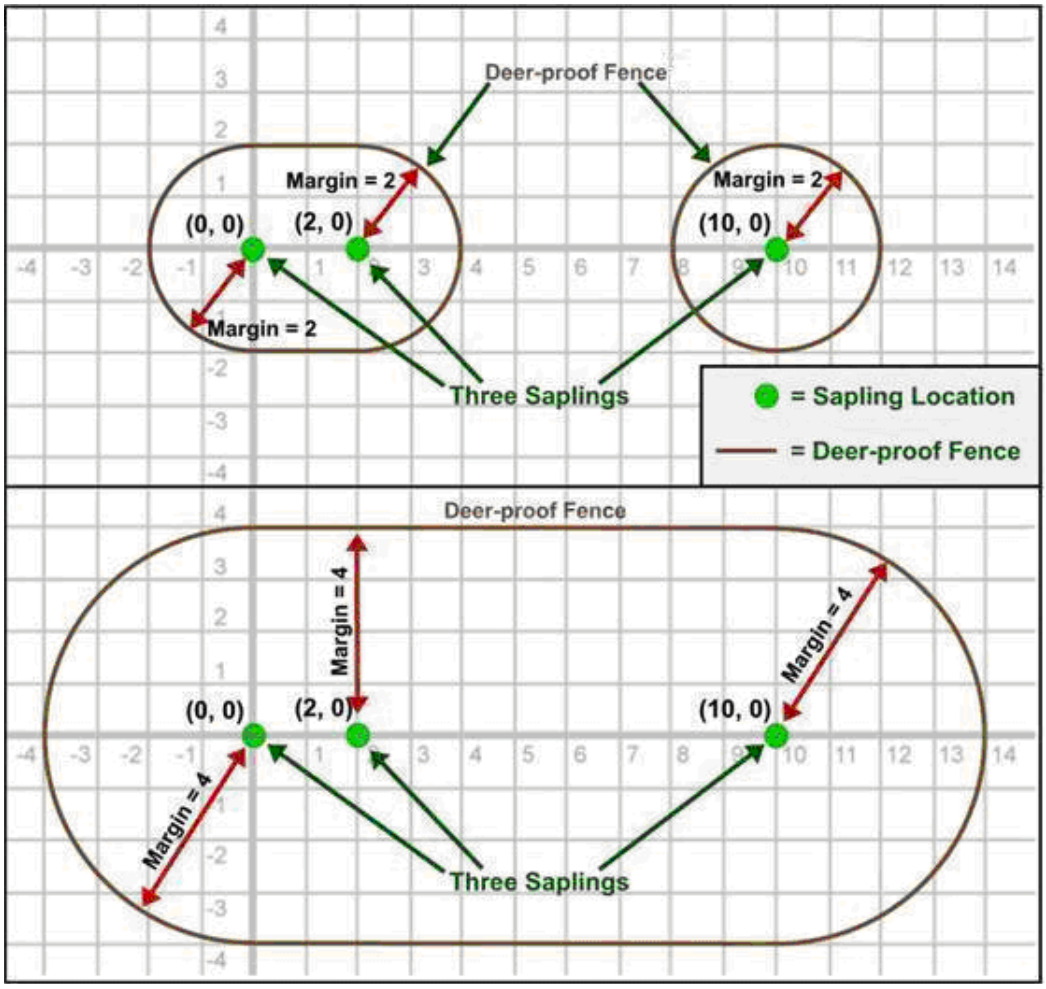

Figure 6: Deer-proof fences.

Input

The input consists of multiple test cases. The first line of each test case contains integers N ($0 < N \leq 9$), which is the number of saplings, and M ($0 < M \leq 200$), which is the margin required around each sapling. This line is followed by N additional lines. Each of these N lines contains two integers x and y that describe the Cartesian coordinates of a sapling ($|x| \leq 100$ and $|y| \leq 100$). No two saplings are in the same location. For simplicity the saplings can all be considered as points and the thickness of deer-proof fences can be considered zero.

The last test case is followed by a line containing two zeros.

Output

For each test case, print the case number (starting with 1) followed by the minimum total length of fencing required to protect the saplings with the given margin. Print the length with two digits to the right of the decimal point. Follow the format of the sample output.

Sample Input	Output for the Sample Input
3 2 0 0 2 0 10 0 3 4 0 0 2 0 10 0 0 0	Case 1: length = 29.13 Case 2: length = 45.13

Problem G
House of Cards
Input file: game.in

Axel and Birgit like to play a card game in which they build a house of cards, gaining (or losing) credits as they add cards to the house. Since they both have very steady hands, the house of cards never collapses. They use half a deck of standard playing cards. A standard deck has four suits, two are red and two are black. Axel and Birgit use only two suits, one red, one black. Each suit has 13 ranks. We use the notation 1R, 2R, …, 13R, 1B, 2B, …, 13B to denote ranks and colors.

The players begin by selecting a subset of the cards, usually all cards of rank up to some maximum value *M*. After shuffling the modified deck, they take eight cards from the top of the deck and place them consecutively from left to right to form four "peaks." For instance, if $M = 13$ and if the first ten cards (from 26) are:

```
6B 3R 5B 2B 1B 5R 13R 7B 11R 1R ...
```

then the game starts off with four peaks and three valleys as shown in Figure 7.

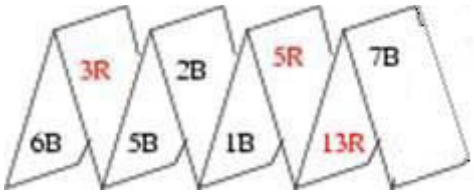

Figure 7: Peaks and valleys formed by the top 8 cards in the deck.

The remaining cards are placed face up, in a single row.

Each player is identified with a color, red or black. Birgit is always black and Axel is always red. The color of the first card used for the peaks and valleys determines which player has the first turn. For the example in Figure 7, Birgit has the first turn since the first card is `6B`.

Players alternate taking turns. A turn consists of removing the card from the front of the row of cards and then doing one of the following:

1. Holding the card until the next turn (this is a "held card").
2. Covering the valley between two peaks with the drawn card or the held card, forming a "floor." The remaining card, if any, is held.
3. Placing two cards over a floor, forming a peak (one of the cards must be a "held" card).

Not all options are always available. At most one card may be held at any time, so the first option is possible only if the player is not already holding a card.

Since the cards in the row are face up, both players know beforehand the order in which the cards are drawn.

If the player forms a downward-pointing triangle by adding a floor, or an upward-pointing triangle by adding a peak, then the scores are updated as follows. The sum of the ranks of the three cards in the triangle is added to the score of the player whose color is the same as the majority of cards in the triangle. If no triangle is formed during a play, both scores remain unchanged.

In the example from Figure 7, if Birgit places her card (`11R`) on the middle valley, she gains 14 points. If she places her card on the left valley, Axel gains 19 points. If she places her card on the right valley, Axel gains 29 points.

If no more cards remain to be drawn at the end of a turn, the game is over. If either player holds a card at this time, the rank of that card is added to (subtracted from) that player's score if the color of the card is the same as (different from) that player's color.

When the game is over, the player with the lower score pays a number of Swedish Kronor (Kronor is the plural of Krona) equal to the difference between the two scores to the other player. In case of a tie there is no pay out.

You must write a program that takes a description of a shuffled deck and one of the players' names and find the highest amount that player can win (or the player's minimum loss), assuming that the other player always makes the best possible moves.

Input

The input file contains multiple test cases representing different games. Each test case consists of a name (either `Axel` or `Birgit`), followed by a maximum rank M ($5 \le M \le 13$), followed by the rank and color of the $2M$ cards in the deck in their shuffled order. Every combination of rank (from 1 through M) and color will appear once in this list. The first eight cards form the initial row of peaks from left to right in the order drawn, and the remaining items show the order of the rest of the cards.

The final test case is followed by a line containing the word `End`.

Output

For each test case, print the test case number (starting with 1), the name of the player for the test case, and the amount that the player wins or loses. If there is a tie, indicate this instead of giving an amount. Follow the sample output format.

Sample Input	Output for the Sample Input
Axel 5 1R 2R 3R 4R 5R 5B 4B 3B 2B 1B Birgit 5 1R 2R 3R 4R 5R 5B 4B 3B 2B 1B Birgit 5 1R 1B 3R 4R 5R 5B 4B 3B 2R 2B End	Case 1: Axel wins 1 Case 2: Birgit loses 1 Case 3: Axel and Birgit tie

Problem H

The Ministers' Major Mess

Input file: major.in

The ministers of the remote country of Stanistan are having severe problems with their decision making. It all started a few weeks ago when a new process for deciding which bills to pass was introduced. This process works as follows. During each voting session, there are several bills to be voted on. Each minister expresses an opinion by voting either "yes" or "no" for some of these bills. Because of limitations in the design of the technical solution used to evaluate the actual voting, each minister may vote on only at most four distinct bills (though this does not tend to be a problem, as most ministers only care about a handful of issues). Then, given these votes, the bills that are accepted are chosen in such a way that each minister gets more than half of his or her opinions satisfied.

As the astute reader has no doubt already realized, this process can lead to various problems. For instance, what if there are several possible choices satisfying all the ministers, or even worse, what if it is impossible to satisfy all the ministers? And even if the ministers' opinions lead to a unique choice, how is that choice found?

Your job is to write a program to help the ministers with some of these issues. Given the ministers' votes, the program must find out whether all the ministers can be satisfied, and if so, determine the decision on those bills for which, given the constraints, there is only one possible choice.

Input

Input consists of multiple test cases. Each test case starts with integers B ($1 \le B \le 100$), which is the number of distinct bills to vote on, and M ($1 \le M \le 500$), which is the number of ministers. The next M lines give the votes of the ministers. Each such line starts with an integer $1 \le k \le 4$, indicating the number of bills that the minister has voted on, followed by the k votes. Each vote is of the format <*bill*> <*vote*>, where <*bill*> is an integer between 1 and B identifying the bill that is voted on, and <*vote*> is either `y` or `n`, indicating that the minister's opinion is "yes" or "no." No minister votes on the same bill more than once. The last test case is followed by a line containing two zeros.

Output

For each test case, print the test case number (starting with 1) followed by the result of the process. If it is impossible to satisfy all ministers, the result should be `impossible`. Otherwise, the result should be a string of length B, where the i^{th} character is `y`, `n`, or `?`, depending on whether the decision on the i^{th} bill should be "yes," whether it should be "no," or whether the given votes do not determine the decision on this bill.

Sample Input	Output for the Sample Input
5 2 4 2 y 5 n 3 n 4 n 4 4 y 3 y 5 n 2 y 4 2 4 1 y 2 y 3 y 4 y 3 1 n 2 n 3 n 0 0	Case 1: ?y??n Case 2: impossible

Problem I
Struts and Springs
Input file: springs.in

Struts and *springs* are devices that determine the way in which rectangular windows on a screen are resized or repositioned when the enclosing window is resized. A window occupies a rectangular region of the screen, and it can enclose other windows to yield a hierarchy of windows. When the outermost window is resized, each of the immediately enclosed windows may change position or size (based on the placement of struts and springs); these changes may then affect the position and/or size of the windows they enclose.

A *strut* is conceptually a fixed length rod placed between the horizontal or vertical edges of a window, or between an edge of a window and the corresponding edge of the immediately enclosing window. When a strut connects the vertical or horizontal edges of a window, then the height or width of that window is fixed. Likewise, when a strut connects an edge of a window to the corresponding edge of the immediately enclosing window, then the distance between those edges is fixed. *Springs*, however, may be compressed or stretched, and may be used in place of struts.

Each window except the outermost has six struts or springs associated with it. One connects the vertical edges of the window, and another connects the horizontal edges of the window. Each of the other four struts or springs connects an edge of the window with the corresponding edge of the enclosing window. The sum of the lengths of the three vertical struts or springs equals the height of the enclosing window; similarly, the sum of the lengths of the three horizontal struts or springs equals the width of the enclosing window. When the enclosing window's width changes, any horizontal springs connected to the window are stretched or compressed in equal proportion, so that the new total length of struts and springs equals the new width. A similar action takes place for a change in the enclosing window's height. If all three vertical or horizontal components are struts, then the top or rightmost strut, respectively, is effectively replaced by a spring.

You must write a program that takes the initial sizes and positions of a set of windows (with one window guaranteed to enclose all others), the placement of struts and springs, and requests to resize the outermost window and then determines the modified size and position of each window for each size request.

Input

Input consists of multiple test cases corresponding to different sets of windows. Each test case begins with a line containing four integers *nwin*, *nresize*, *owidth*, and *oheight*. *nwin* is the number of windows (excluding the outer window which encloses all others), *nresize* is the number of outer window resize requests, and *owidth* and *oheight* are the initial width and height of the outer window.

Each of the next *nwin* lines contains 10 non-negative integers and describes a window. The first two integers give the initial x and y displacement of the upper left corner of the window with respect to the upper left corner of the outermost window. The next two integers give the initial width and height of the window. Each of the final six integers is either 0 (for a strut) or 1 (for a spring). The first two specify whether a strut or spring connects the vertical and horizontal edges of the window respectively, and the last four specify whether a strut or spring connects the tops, bottoms, left sides and right sides of the window and its immediately enclosing window.

Each of the last *nresize* lines in a test gives a new width and height for the outermost window – a resize operation. For each of these, your program must determine the size and placement of each of the *nwin* inner windows. The test data is such that, after every resizing operation, every strut and spring has a positive integral length, and different window boundaries do not touch. Also, resizing never causes one window to jump inside another.

There are at most 100 windows and 100 resize operations in a test case, and the outermost window's width and height never exceed 1,000,000. The last test case is followed by a line with four zeros.

Output

For each resize operation in a test case, print the test case number (starting with 1) followed by the resize operation number (1, 2, …) on a line by itself. Then on each of the next *nwin* lines, print the window number, position (*x* and *y*) and size (width and height) of each of the inner windows of the test case as a result of resizing the outer window. Windows in each test case are numbered sequentially (1, 2, …) to match their position in the input, and should be output in that order. Follow the format shown in the sample output.

Sample Input	Output for the Sample Input

1 1 50 100
10 10 30 10 1 0 0 0 0 0
70 150
2 1 50 100
10 10 30 10 1 0 0 0 0 0
10 80 20 10 1 1 0 0 0 0
70 150
1 2 60 60
10 10 20 30 1 0 1 1 1 1
90 90
120 120
0 0 0 0
	Case 1, resize operation 1:
	    Window 1, x = 10, y = 60, width = 50, height = 10
	Case 2, resize operation 1:
	    Window 1, x = 10, y = 60, width = 50, height = 10
	    Window 2, x = 10, y = 80, width = 40, height = 60
	Case 3, resize operation 1:
	    Window 1, x = 15, y = 20, width = 30, height = 30
	Case 3, resize operation 2:
	    Window 1, x = 20, y = 30, width = 40, height = 30

Problem J
Subway Timing
Input: subway.in

Like most modern cities, Stockholm has a well-developed public transportation system. The heart of public transportation in Stockholm is the subway. A topological map of the subway system illustrates the different subway lines and how they are connected, as illustrated in Figure 8. For this problem you should assume that a subway map is always tree-shaped, even though this is not quite true for Stockholm because of the cycle formed by the green and blue lines.

Figure 8: Stockholm subway map

A topological map says very little about the geometry of a subway system, such as the distances (and consequently travel times) between different subway stations. For instance, as most students in Stockholm know, the distance between "*Tekniska Högskolan*" (The Royal Institute of Technology) and "*Universitetet*" (Stockholm University) is quite large, even though there is no indication of this on the map.

You must write a program that can augment a topological map by writing down the time required to travel between every pair of adjacent subway stations. Fortunately, those travel times are known, so you do not have to measure the times yourself. But the actual travel times are given in seconds, while the times must be written on the map as estimates of integral numbers of minutes.

A natural way of estimating times might be to simply round all the travel times to the nearest minute. However, this can cause huge cumulative errors. For instance, in the Stockholm map, this estimation method could result in an error as large as 15 minutes in the total travel time between some pairs of stations. In order to counter this, your program may choose to round some travel times up and round other travel times down. The rounding must be done in such a way that the largest cumulative error between any pair of stations is minimized.

Input

The input consists of several test cases. Each test case starts with an integer N ($1 \le N \le 100$), which is the number of subway stations. The N stations are identified using the integers from 1 to N. Each of the next N-1 lines contains three integers a, b and t ($1 \le a, b \le N$, and $1 \le t \le 300$), indicating that stations a and b are adjacent and that it takes t seconds to travel between them. For simplicity, ignore the time a train spends standing still at a station.

The last test case is followed by the integer zero on a line by itself.

Output

For each test case, print the case number (starting with 1) then the largest rounding error in seconds of travel time between any pair of stations when the times for adjacent pairs of stations are rounded optimally. Follow the format of the sample output.

Sample Input	Output for the Sample Input
2 1 2 110 4 1 2 40 2 3 40 3 4 40 4 1 2 90 1 3 90 1 4 90 0	Case 1: 10 Case 2: 40 Case 3: 60

Problem K
Suffix-Replacement Grammars
Input file: suffix.in

As computer programmers, you have likely heard about regular expressions and context-free grammars. These are rich ways of generating sets of strings over a small alphabet (otherwise known as a formal language). There are other, more esoteric ways of generating languages, such as tree-adjoining grammars, context-sensitive grammars, and unrestricted grammars. This problem uses a new method for generating a language: a suffix-replacement grammar.

A suffix-replacement grammar consists of a starting string *S* and a set of suffix-replacement rules. Each rule is of the form *X* → *Y*, where *X* and *Y* are equal-length strings of alphanumeric characters. This rule means that if the suffix (that is, the rightmost characters) of your current string is *X*, you can replace that suffix with *Y*. These rules may be applied arbitrarily many times.

For example, suppose there are 4 rules A → B, AB → BA, AA → CC, and CC → BB. You can then transform the string AA to BB using three rule applications: AA → AB (using the A → B rule), then AB → BA (using the AB → BA rule), and finally BA → BB (using the A→ B rule again). But you can also do the transformation more quickly by applying only 2 rules: AA → CC and then CC → BB.

You must write a program that takes a suffix-replacement grammar and a string *T* and determines whether the grammar's starting string *S* can be transformed into the string *T*. If this is possible, the program must also find the minimal number of rule applications required to do the transformation.

Input
The input consists of several test cases. Each case starts with a line containing two equal-length alphanumeric strings *S* and *T* (each between 1 and 20 characters long, and separated by whitespace), and an integer *NR* ($0 \le NR \le 100$), which is the number of rules. Each of the next *NR* lines contains two equal-length alphanumeric strings *X* and *Y* (each between 1 and 20 characters long, and separated by whitespace), indicating that *X* → *Y* is a rule of the grammar. All strings are case-sensitive. The last test case is followed by a line containing a period.

Output
For each test case, print the case number (beginning with 1) followed by the minimum number of rule applications required to transform *S* to *T*. If the transformation is not possible, print `No solution`. Follow the format of the sample output.

Sample Input	Output for the Sample Input
AA BB 4 A B AB BA AA CC CC BB A B 3 A C B C c B .	Case 1: 2 Case 2: No solution