Problem A
Spreadsheet Calculator

A spreadsheet is a rectangular array of cells. Cells contain data or expressions that can be evaluated to obtain data. A "simple" spreadsheet is one in which data are integers and expressions are mixed sums and differences of integers and cell references. For any expression, if each cell that is referenced contains an integer, then the expression can be replaced by the integer to which the expression evaluates. You are to write a program which evaluates simple spreadsheets.

Input

Input consists of a sequence of simple spreadsheets. Each spreadsheet begins with a line specifying the number of rows and the number of columns. No spreadsheet contains more than 20 rows or 10 columns. Rows are labeled by capital letters A through T. Columns are labeled by decimal digits 0 through 9. Therefore, the cell in the first row and first column is referenced as A0; the cell in the twentieth row and fifth column is referenced as T4.

Following the specification of the number of rows and columns is one line of data for each cell, presented in row-major order. (That is, all cells for the first row come first, followed by all cells for the second row, etc.) Each cell initially contains a signed integer value or an expression involving unsigned integer constants, cell references, and the operators + (addition) and – (subtraction). If a cell initially contains a signed integer, the corresponding input line will begin with an optional minus sign followed by one or more decimal digits. If a cell initially contains an expression, its input line will contain one or more cell references or unsigned integer constants separated from each other by + and – signs. Such a line must begin with a cell reference. No expression contains more than 75 characters. No line of input contains leading blanks. No expression contains any embedded blanks. However, any line may contain trailing blanks.

The end of the sequence of spreadsheets is marked by a line specifying 0 rows and 0 columns.

Output

For each spreadsheet in the input, you are to determine the value of each expression and display the resulting spreadsheet as a rectangular array of numbers with the rows and columns appropriately labeled. In each display, all numbers for a column must appear right-justified and aligned with the column label. Operators are evaluated left to right in each expression; values in cells are always less than 10000 in absolute value. Since expressions may reference cells that themselves contain expressions, the order in which cells are evaluated is dependent on the expressions themselves.

If one or more cells in a spreadsheet contain expressions with circular references, then the output for that spreadsheet should contain only a list of the unevaluated cells in row-major order, one per line, with each line containing the cell label, a colon, a blank, and the cell's original expression.

A blank line should appear following the output for each spreadsheet. Sample input and output are below.

Sample Input

```
2 2
A1+B1
5
3
B0-A1
3 2
A0
5
C1
7
A1+B1
```

Output for the Sample Input

```
      0     1
A     3     5
B     3    -2

A0: A0
B0: C1
C1: B0+A1
```

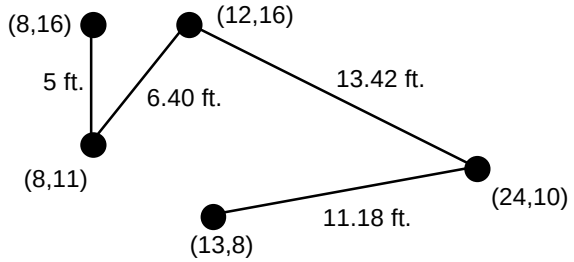
continued on reverse side ∅

B0+A1

0 0

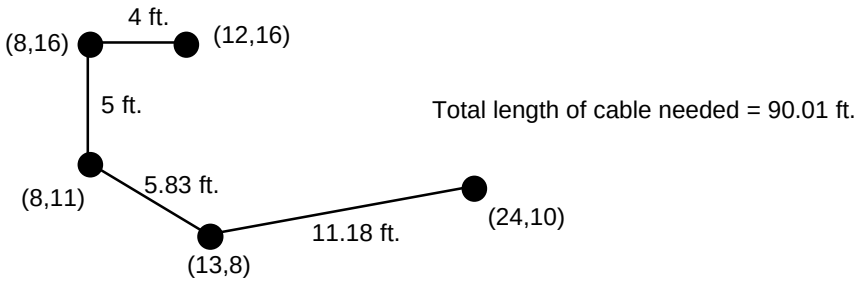
Problem B
Getting in Line

Computer networking requires that the computers in the network be linked. This problem considers a “linear” network in which the computers are chained together so that each is connected to exactly two others except for the two computers on the ends of the chain which are connected to only one other computer. A picture is shown below. Here the computers are the black dots and their locations in the network are identified by planar coordinates (relative to a coordinate system not shown in the picture). Distances between linked computers in the network are shown in feet.



For various reasons it is desirable to minimize the length of cable used. Your problem is to determine how the computers *should be* connected into such a chain to minimize the total amount of cable needed. In the installation being constructed, the cabling will run beneath the floor, so the amount of cable used to join 2 adjacent computers on the network will be equal to the distance between the computers plus 16 additional feet of cable to connect from the floor to the computers and provide some slack for ease of installation.

The picture below shows the optimal way of connecting the computers shown above, and the total length of cable required for this configuration is (4+16)+ (5+16) + (5.83+16) + (11.18+16) = 90.01 feet.



Input

The input file will consist of a series of data sets. Each data set will begin with a line consisting of a single number indicating the number of computers in a network. Each network has at least 2 and at most 8 computers. A value of 0 for the number of computers indicates the end of input. After the initial line in a data set specifying the number of computers in a network, each additional line in the data set will give the coordinates of a computer in the network. These coordinates will be integers in the range 0 to 150. No two computers are at identical locations and each computer will be listed once.

Output

The output for each network should include a line which tells the number of the network (as determined by its position in the input data), and one line for each length of cable to be cut to connect each adjacent pair of computers in the network. The final line should be a sentence indicating the total amount of cable used. **In listing the lengths of cable to be cut, traverse the network from one end to the other**. (It makes no difference at which end you start.) Use a format similar to the one shown in the sample output, with a line of asterisks separating output for different networks and with distances in feet printed to 2 decimal places.

Sample Input

```
6
5 19
55 28
38 101
28 62
111 84
43 116
5
11 27
84 99
142 81
88 30
95 38
3
132 73
49 86
72 111
0
```

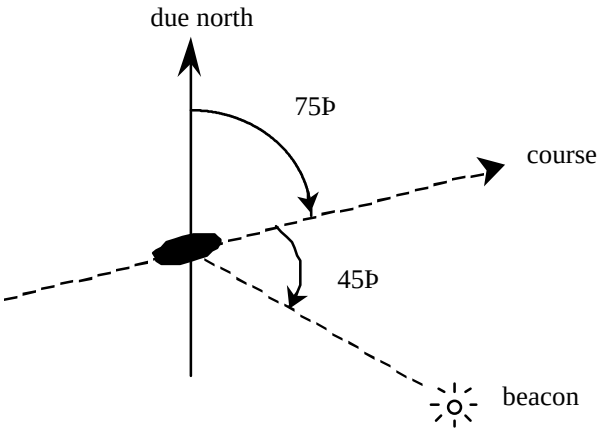
Output for the Sample Input

```
**********************************************************
Network #1
Cable requirement to connect (5,19) to (55,28) is 66.80 feet.
Cable requirement to connect (55,28) to (28,62) is 59.42 feet.
Cable requirement to connect (28,62) to (38,101) is 56.26 feet.
Cable requirement to connect (38,101) to (43,116) is 31.81 feet.
Cable requirement to connect (43,116) to (111,84) is 91.15 feet.
Number of feet of cable required is 305.45.
**********************************************************
Network #2
Cable requirement to connect (11,27) to (88,30) is 93.06 feet.
Cable requirement to connect (88,30) to (95,38) is 26.63 feet.
Cable requirement to connect (95,38) to (84,99) is 77.98 feet.
Cable requirement to connect (84,99) to (142,81) is 76.73 feet.
Number of feet of cable required is 274.40.
**********************************************************
Network #3
Cable requirement to connect (132,73) to (72,111) is 87.02 feet.
Cable requirement to connect (72,111) to (49,86) is 49.97 feet.
Number of feet of cable required is 136.99.
```

Problem C
Radio Direction Finder

A boat with a directional antenna can determine its present position with the help of readings from local beacons. Each beacon is located at a known position and emits a unique signal. When a boat detects a signal, it rotates its antenna until the signal is at maximal strength. This gives a relative bearing to the position of the beacon. Given a previous beacon reading (the time, the relative bearing, and the position of the beacon), a new beacon reading is usually sufficient to determine the boat's present position. You are to write a program to determine, when possible, boat positions from pairs of beacon readings.

For this problem, the positions of beacons and boats are relative to a rectangular coordinate system. The positive *x*-axis points east; the positive *y*-axis points north. The course is the direction of travel of the boat and is measured in degrees clockwise from north. That is, north is 0°, east is 90°, south is 180°, and west is 270°. The relative bearing of a beacon is given in degrees clockwise relative to the course of the boat. A boat's antenna cannot indicate on which side the beacon is located. A relative bearing of 90° means that the beacon is toward 90° or 270°.



The boat's course is 75°.
The beacon has a relative bearing of 45Þ from the boat's course.

Input

The first line of input is an integer specifying the number of beacons (at most 30). Following that is a line for each beacon. Each of those lines begins with the beacon's name (a string of 20 or fewer alphabetic characters), the *x*-coordinate of its position, and the *y*-coordinate of its position. These fields are single-space separated.

Coming after the lines of beacon information is an integer specifying a number of boat scenarios to follow. A boat scenario consists of three lines, one for velocity and two for beacon readings.

Data on input line	Meaning of data
course speed	the boat's course, the speed at which it is traveling
time#1 name#1 angle#1	time of first beacon reading, name of first beacon, relative bearing of first beacon
time#2 name#2 angle#2	time of second reading, name of second beacon, relative bearing of second beacon

All times are given in minutes since midnight measured over a single 24-hour period. The speed is the distance (in units matching those on the rectangular coordinate system) over time. The second line of a scenario gives the first beacon reading as the time of the reading (an integer), the name of the beacon, and the angle of the reading as measured from the boat's course. These 3 fields have single space separators. The third line gives the second beacon reading. The time for that reading will always be at least as large as the time for the first reading.

Output

For each scenario, your program should print the scenario number (Scenario 1, Scenario 2, etc.) and a message indicating the position (rounded to 2 decimal places) of the boat as of the time of the *second* beacon reading. If it is impossible to determine the position of the boat, the message should say "`Position cannot be determined`." Sample input and corresponding correct output are shown below.

Sample Input

```
4
First 2.0 4.0
Second 6.0 2.0
Third 6.0 7.0
Fourth 10.0 5.0
2
0.0 1.0
1 First 270.0
2 Fourth 90.0
116.5651 2.2361
4 Third 126.8699
5 First 319.3987
```

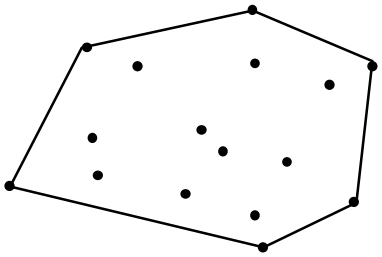
Output for the Sample Input

```
Scenario 1: Position cannot be determined
Scenario 2: Position is (6.00, 5.00)
```

Problem D
Moth Eradication

Entomologists in the Northeast have set out traps to determine the influx of Jolliet moths into the area. They plan to study eradication programs that have some potential to control the spread of the moth population.

The study calls for organizing the traps in which moths have been caught into compact regions, which will then be used to test each eradication program. A region is defined as the polygon with the minimum length perimeter that can enclose all traps within that region. For example, the traps (represented by dots) of a particular region and its associated polygon are illustrated below.



You must write a program that can take as input the locations of traps in a region and output the locations of traps that lie on the perimeter of the region as well as the length of the perimeter.

Input

The input file will contain records of data for several regions. The first line of each record contains the number (an integer) of traps for that region. Subsequent lines of the record contain 2 real numbers that are the *x*- and *y*-coordinates of the trap locations. Data within a single record will not be duplicated. End of input is indicated by a region with 0 traps.

Output

Output for a single region is displayed on at least 3 lines:

First line:	The number of the region. (The first record corresponds to region #1, the second to region #2, etc.)
Next line(s):	A listing of all the points that appear on the perimeter of the region. The points must be identified in the standard form "(*x*-coordinate,*y*-coordinate)" rounded to a single decimal place. The starting point for this listing is irrelevant, but the listing must be oriented *clockwise* and *begin and end with the same point*. For collinear points, any order which describes the minimum length perimeter is acceptable.
Last line:	The length of the perimeter of the region rounded to 2 decimal places.

One blank line must separate output from consecutive input records.

A sample input file with records for 3 regions followed by correct output for the sample input is shown on the reverse.

Sample Input

```
3
1 2
4 10
5 12.3
6
0 0
1 1
3.1 1.3
3 4.5
6 2.1
2 -3.2
7
1 0.5
5 0
4 1.5
3 -0.2
2.5 -1.5
0 0
2 2
0
```

Output for the Sample Input

```
Region #1:
(1.0,2.0)-(4.0,10.0)-(5.0,12.3)-(1.0,2.0)
Perimeter length = 22.10

Region #2:
(0.0,0.0)-(3.0,4.5)-(6.0,2.1)-(2.0,-3.2)-(0.0,0.0)
Perimeter length = 19.66

Region #3:
(0.0,0.0)-(2.0,2.0)-(4.0,1.5)-(5.0,0.0)-(2.5,-1.5)-(0.0,0.0)
Perimeter length = 12.52
```

Problem E
Department of Redundancy Department

When designing tables for a relational database, a functional dependency (FD) is used to express the relationship between the different fields. A functional dependency is concerned with the relationship of values of one set of fields to those of another set of fields. The notation `X->Y` is used to denote that when supplied values to the field(s) in set `X`, the assigned value for each field in set `Y` can be determined. For example, if a database table is to contain fields for the *social security number* (`S`), *name* (`N`), *address* (`A`), and *phone* (`P`) and each person has been assigned a unique value for `S`, the `S` field functionally determines the `N`, `A` and `P` fields. This is written as `S->NAP`.

Develop a program that will identify each redundant FD in each input group of FDs. An FD is redundant if it can be derived using other FDs in the group. For example, if the group contains the FDs `A->B`, `B->C`, and `A->C`, then the third FD is redundant since the field set `C` can be derived using the first two. (The `A` fields determine values for the `B` fields, which in turn determine values for the fields in `C`.) In the group `A->B`, `B->C`, `C->A`, `A->C`, `C->B`, and `B->A`, all the FDs are redundant.

Input

The input file contains an arbitrary number of groups of FDs. Each group is preceded by a line containing an integer no larger than 100 specifying the number of FDs in that group. A group with zero FDs indicates the end of the input. Each FD in the group appears on a separate line containing two non-empty lists of field names separated by the characters `-` and `>`. The lists of field names contain only uppercase alphabetic characters. Functional dependency lines contain no blanks or tabs. There are no trivially redundant FDs (for example, `A->A`). For identification purposes, groups are numbered sequentially, starting with 1; the FDs are also numbered sequentially, starting with 1 in each group.

Output

For each group, in order, your program must identify the group, each redundant FD in the group, and a sequence of the other FDs in the group which were used to determine the indicated FD is redundant. If more than one sequence of FDs can be used to show another FD is redundant, any such sequence is acceptable, even if it is not the shortest proof sequence. Each FD in an acceptable proof sequence must, however, be necessary. If a group of FDs contains no redundancy, display `No redundant FDs.`

Sample Input

```
3
A->BD
BD->C
A->C
6
P->RST
VRT->SQP
PS->T
Q->TR
```

Output for the Sample Input

```
Set number 1
     FD 3 is redundant using FDs: 1 2

Set number 2
     FD 3 is redundant using FDs: 1
     FD 5 is redundant using FDs: 4 6 2

Set number 3
     FD 5 is redundant using FDs: 1 3
```

QS->P
SR->V
5
A->B
A->C
B->D
C->D
A->D
3
A->B
B->C
A->D
0

--OR--

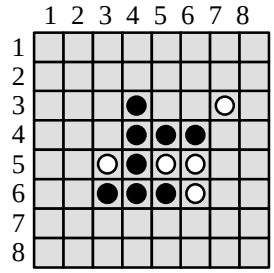
FD 5 is redundant using FDs: 2 4

Set number 4

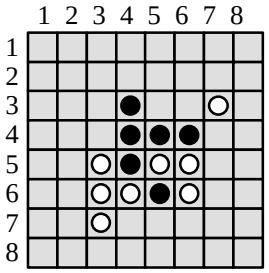
No redundant FDs.

Problem F
Othello

Othello is a game played by two people on an 8 x 8 board, using disks that are white on one side and black on the other. One player places disks with the white side up and the other player places disks with the black side up. The players alternate placing one disk on an unoccupied space on the board. In placing a disk, the player **must** bracket at least one of the other color disks. Disks are bracketed if they are in a straight line horizontally, vertically, or diagonally, with a disk of the current player's color at each end of the line. When a move is made, **all** the disks that were bracketed are changed to the color of the player making the move. (It is possible that disks will be bracketed across more than one line in a single move.)



Legal Moves for White
(2,3), (3,3), (3,5), (3,6),
(6,2), (7,3), (7,4), (7,5)



Board Configuration after
White Moves to (7,3)

Write a program to read a series of Othello games. The first line of the input is the number of games to be processed. Each game consists of a board configuration followed by a list of commands. The board configuration consists of 9 lines. The first 8 specify the current state of the board. Each of these 8 lines contains 8 characters, and each of these characters will be one of the following:

'`-`' indicating an unoccupied square
'`B`' indicating a square occupied by a black disk
'`W`' indicating a square occupied by a white disk

The ninth line is either a '`B`' or a '`W`' to indicate which is the current player. You may assume that the data is legally formatted.

The commands are to list all possible moves for the current player, make a move, or quit the current game. There is one command per line with no blanks in the input. Commands are formatted as follows:

List all possible moves for the current player. The command is an '`L`' in the first column of the line. The program should go through the board and print all legal moves for the current player in the format (*x*,*y*) where *x* represents the row of the legal move and *y* represents its column. These moves should be printed in row major order which means:

1) all legal moves in row number *i* will be printed before any legal move in row number *j* if *j* is greater than *i*

and 2) if there is more than one legal move in row number *i*, the moves will be printed in ascending order based on column number.

All legal moves should be put on one line. If there is no legal move because it is impossible for the current player to bracket any pieces, the program should print the message "`No legal move`."

Make a move. The command is an '`M`' in the first column of the line, followed by 2 digits in the second and third column of the line. The digits are the row and the column of the space to place the piece of the current player's color, *unless the current player has no legal move*. If the current player has no legal move, the current player is first changed to the other player and the move will be the move of the new current player. You may assume that the move is then legal. You should record the changes to the board, including adding the new piece and changing the color of all bracketed pieces. At the end of the move, print the number of pieces of each color on the board in the format "`Black - xx  White - yy`" where `xx` is the number of black pieces on the board and `yy` is the number of white pieces on the board. After a move, the current player will be changed to the player that did not move.

Quit the current game. The command will be a ‘`Q`’ in the first column of the line. At this point, print the final board configuration using the same format as was used in the input. This terminates input for the current game.

You may assume that the commands will be syntactically correct. Put one blank line between output from separate games and no blank lines anywhere else in the output.

Sample input

```
2
--------
--------
--------
---WB---
---BW---
--------
--------
--------
W
L
M35
L
Q
WWWWB---
WWWB----
WWB-----
WB------
--------
--------
--------
--------
B
L
M25
L
Q
```

Output for the Sample Input

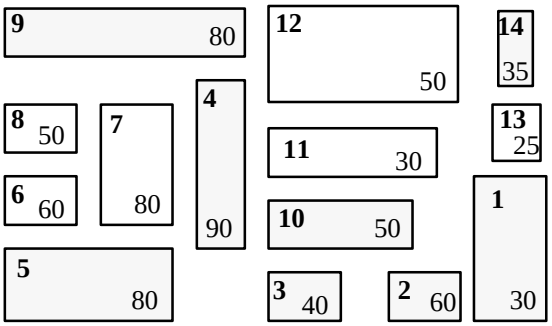
```
(3,5) (4,6) (5,3) (6,4)
Black -  1 White -  4
(3,4) (3,6) (5,6)
--------
--------
----W---
---WW---
---BW---
--------
--------
--------

No legal move.
Black -  3 White - 12
(3,5)
WWWWB---
WWWWW---
WWB-----
WB------
--------
--------
--------
--------
```

1992 ACM Scholastic Programming Contest Finals
sponsored by AT&T EasyLink Services

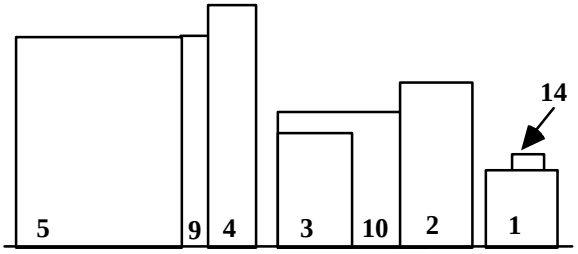
Problem G
Urban Elevations

An elevation of a collection of buildings is an orthogonal projection of the buildings onto a vertical plane. An external elevation of a city would show the skyline and the faces of the "visible" buildings of the city as viewed from outside the city from a certain direction. A southern elevation shows no sides; it shows the perfectly rectangular faces of buildings or parts of faces of buildings not obstructed on the south by taller buildings. For this problem, you must write a program that determines which buildings of a city are visible in a southern elevation.

For simplicity, assume all the buildings for the elevation are perfect rectangular solids, each with two sides that run directly east-west and two running directly north-south. Your program will find the buildings that appear in a southern elevation based on knowing the positions and heights of each city building. That data can be illustrated by a map of the city as in the diagram on the left below. The southern elevation for that city is illustrated in the diagram on the right.



City map. Boldface numbers (in the upper left of each building) identify the buildings.
Plain numbers (lower right) are the building heights.
The shaded



Southern Elevation

Input

Input for your program consists of the numeric description of maps of several cities. The first line of each map contains the number of buildings in the city (a non-negative integer less than 101). Each subsequent line of a map contains data for a single building — 5 real numbers separated by spaces in the following order:

x-coordinate of the southwest corner
y-coordinate of the southwest corner
width of the building (length of the south side)
depth of the building (length of the west side)
height of the building

Each map is oriented on a rectangular coordinate system so that the positive *x*-axis points east and the positive *y*-axis points north. Assume that all input for each map corresponds to a legitimate map (the number of buildings is the same as the number of subsequent lines of input for the map; no two buildings in a single map overlap). Input is terminated by the number 0 representing a map with no buildings.

Output

Buildings are numbered according to where their data lines appear in the map's input data — building #1 corresponding to the first line of building data, building #2 data to the next line, and building #*n* to the *n*th line of building data for that map. (Buildings on subsequent maps also begin their numbering with 1.)

For each map, output begins with line identifying the map (map #1, map #2, etc.) On the next line the numbers of the visible buildings as they appear in the southern elevation, ordered south-to-north, west-to-east. This means that if building *n* and building *m* are visible buildings and if the southwest corner of building *n* is west of the southwest corner of building *m*, then number *n* is printed before number *m*. If building *n* and building *m* have the same *x*-coordinate for their southwest corners and if building *n* is south of building *m*, then the number *n* is printed before the number *m*. For this program, a building is considered visible whenever the part of its southern face that appears in the elevation has strictly positive area. One blank line must separate output from consecutive input records.

Sample Input

```
14
160 0 30 60 30
125 0 32 28 60
95 0 27 28 40
70 35 19 55 90
0 0 60 35 80
0 40 29 20 60
35 40 25 45 80
0 67 25 20 50
0 92 90 20 80
95 38 55 12 50
95 60 60 13 30
95 80 45 25 50
165 65 15 15 25
165 85 10 15 35
0
```

Output for the Sample Input

```
For map #1, the visible buildings are numbered as follows:
5 9 4 3 10 2 1 14
```