Wacky World sure is a crazy place! Just ask one of its residents, Walter Winters (his friends call him Wally). You see, Wacky World is a two dimensional world. In Wacky World, there are many different zones. Each zone can be drawn on a map as a grid of squares, with each square being indicated by a coordinate composed of a letter and a number, like A9 or B3 (see example below). The letter indicates the particular column and the number of the particular row that the visitor is in. Wally knows, and you should too, that Wacky World laws only allow the movement of an individual from one square to another in either a horizontal manner or a vertical manner. No diagonal moves are allowed!

But, there is one little wrinkle to Wacky World’s geography. You see, when Wally gets to the edge of a zone, he can instantly teleport to the exact opposite side of the zone without moving an inch! Furthermore, if Wally reaches a corner of a zone, Wally can instantly teleport to any other corner in the zone – again, not moving an inch!

You see, our friend Wally is a bit out of shape and wants to get from one location to another in Wacky World by traveling as little distance as possible. To help him, just figure out for Wally the shortest distance (in squares) that he has to travel to get from his starting location to his ending location. Movement from one location on the Wacky World grid to another adjacent location counts as having moved one square. But remember, it is possible to teleport to opposite sides of Wacky World once you reach the edge of the map without having traveled any squares.

Wally knows this is all so confusing, so he’s figured out a couple of examples ahead of time for you to consider. Let’s say that Wally is in a zone of Wacky World that has dimensions 6 x 6 (that is, six rows and six columns). Then a map of that zone would look like this (notice the way the grid is labeled):

```
1 2 3 4 5 6
A  Y
B
C
D
E
F
```

Now let’s say that Wally wants to get from location X (square E4) to location Y (square B2). The shortest distance Wally has to travel to get from X (E4) to Y (B2) is four squares. How so? Note that from E4, Wally first can travel to F4. That’s one square. Now, since he is at the edge of the zone, he can teleport to the opposite side of the zone, to A4. Remember, at this point he has still only traveled one square. Wally travels no distance during the teleportation process. Next, Wally moves to B4. That brings our total to two squares. Finally, Wally moves to B3 and then to B2, bringing the total distance traveled to four squares. Man, this place really is Wacky!
In this example, you may notice that there are several ways to travel from $x$ to $y$ by traveling only four squares. Your job is merely to report what is the least amount distance that Wally has to travel to get from one location to another. Ready?

**Details of the input**
There will be multiple problem instances, each corresponding to a distinct zone that Wally is traveling in. Each input set will begin with a line of two integers $r$ $c$ ($r$, $c \leq 26$), indicating the size of the zone where $r$ is the number of rows and $c$ is the number of columns. Values of $r = 0$ and $c = 0$ indicates the end of input. The next line will be an integer $n$ ($n \leq 100$) indicating the number of test cases for the particular zone. Finally, there will be $n$ lines of coordinates of the form start finish, where start is Wally's starting location and finish is Wally's final destination.

**Details of the output**
Each result should be titled in the output as follows:

Results for Wacky World – Zone #$k$

where $k$ is the number of the problem instance starting with 1. Following this title should be $n$ lines indicating the results for each test case of the form:

The shortest way to get from start to finish is $x$ squares.

where start and finish are the starting and ending coordinates, respectively, and where $x$ is the shortest distance between those two locations.

Finally, there should be an extra blank line after each output set.

**Sample Input**

```
7 7
7
G1 A7
B2 B3
A1 G7
G1 G7
A2 F3
C6 D4
E3 A2

26 26
4
E3 F25
A1 G26
G7 E13
Q3 E17

0 0
```
Sample Output
Results for Wacky World - Zone #1
The shortest way to get from G1 to A7 is 0 squares.
The shortest way to get from B2 to B3 is 1 squares.
The shortest way to get from A1 to G7 is 0 squares.
The shortest way to get from G1 to G7 is 0 squares.
The shortest way to get from A2 to F3 is 2 squares.
The shortest way to get from C6 to D4 is 3 squares.
The shortest way to get from E3 to A2 is 3 squares.

Results for Wacky World - Zone #2
The shortest way to get from E3 to F25 is 4 squares.
The shortest way to get from A1 to G26 is 6 squares.
The shortest way to get from G7 to E13 is 8 squares.
The shortest way to get from Q3 to E17 is 23 squares.
The Eccentric Time Traveler

Leonard A. Beardman is a time traveler from 1932 who, for some strange reason, chose to visit our present time. Being the curious scientist that he is, Beardman wants to get up to speed with our current technology. After listening to an iPod, he is instantly captivated by the popular MP3 player. Although brilliant, Beardman is also quite an eccentric fellow. He now plans on returning to 1932 and creating his own MP3 player, the iBeard. However, Beardman insists that the MP3 files be stored on “punch cards”—the storage medium of choice in 1932. A punch card is a rigid piece of paper (roughly the size of a dollar bill) that contains 80 columns with 12 punch locations each, for a total of 960 possible locations to be punched. A stack of 143 of these punch cards is 25 millimeters tall. For his iBeard, Beardman will treat each punch location as a binary digit (or “bit”) in which to store an MP3 file. To convince Beardman that this idea is insane, you must show him that the stack produced by storing an MP3 file on punch cards would far exceed the definition of “portable audio.” For example, an MP3 of Led Zeppelin’s Stairway to Heaven encoded at 192kbps is 11 megabytes in size. This means it would take a stack of cards 16,804 millimeters tall (about 55 feet!) to store that MP3 file.

1 megabyte = 1,024 kilobytes = 1,048,576 bytes = 8,388,608 bits

Details of the Input
The first line of input will contain an integer, \( n \), representing the number of MP3 files that Beardman wants to store. The following \( n \) lines of input represent the individual file size—in megabytes—of each MP3. Each size will be an integer.

Details of the Output
There will be one line of output for each MP3 file to be stored. The format is as follows:

The resulting stack is \( x \)mm tall.

where \( x \) is an integer corresponding to the height of the stack. Any fractional part of a millimeter is ignored (e.g. 35.8mm would print as 35mm).

Sample Input
5
11
7
5
9
3

Sample Output
The resulting stack is 16804mm tall.
The resulting stack is 10693mm tall.
The resulting stack is 7638mm tall.
The resulting stack is 13748mm tall.
The resulting stack is 4582mm tall.

Reference:
http://www.cs.uiowa.edu/~jones/cards/history.htm
Oh Draughts!

Checkers, or Draughts, is a common two-player board game. In the most common English variant, each player starts with 12 pieces on their side of an 8x8 square board, as pictured above. The top is player one’s home side, while the bottom is player two’s home side. Pieces may only be on the dark squares of the board.

A normal piece may advance by moving diagonally one position at a time in the direction of the opponent’s side. If an opposing piece sits diagonally adjacent in a player’s normal direction of advancement, but the following diagonal space along the same line is vacant, then the player may “jump” the adjacent piece and land on the vacant space, removing the opponent’s jumped piece from the board in the process. This is called a “capture move.” The objective of checkers is to capture all of the opponent’s pieces.

If a piece reaches the opponent’s side of the board, it is “crowned”, and is considered a “King.” Kings can move in any of the four diagonal directions (as opposed to the normal pieces which must advance only towards the opponent’s home side).

A move in checkers is defined as a given player choosing a piece to advance by one position or jumping an opponent’s diagonally adjacent piece. In English Checkers, players can “double jump” in the same turn by repeatedly jumping opponent’s pieces into vacant squares. For our purposes, we will limit a turn to a single jump only. Also, we will only consider boards where pieces will have at most one possible capture move available.

Your task is to take a draught configuration and determine all possible capture moves currently available to both players.

Details of the Input

The first line of the input indicates the number of games to examine. For each game, the first line contains one positive integer, \( k \) (no greater than 12), representing the number of pieces player one has on the board. The next (potentially long) line contains \( k \) 3-tuples representing the coordinates of each of player one’s pieces in the form “row col type”, where row and col are integers between 1 and 8 corresponding to the image above, and type is the character ‘k’ if the piece has been crowned, or ‘n’ if it has not. For example, the piece in the top right of the board shown above would be listed as “1 8 n”. The same two line pattern repeats for player two’s pieces.
Details of the Output
For each game, the message

Player \( k \):

is printed for each of the two players. On the following line(s), all possible moves are listed in the form

\[ \text{[square x]} \to \text{[square y]}, \text{captures[square z]} \].

where \([\text{square i}]\) is the ordered pair \((r, c)\) representing the row and column of the appropriate piece. There is a space after the comma in the ordered pair.

If more than 1 of a player’s pieces has a capture move available, list all such moves on individual lines, ordered by increasing row of the attacking piece, ties broken by increasing column of the attacking piece (e.g. how you read text, left→right and top→bottom). If no captures are possible, output “No pieces can be captured.” A blank line appears between games.

Sample Input

3

12
1 2 n 1 4 n 1 6 n 1 8 n 2 1 n 2 3 n 2 5 n 2 7 n 3 2 n 3 4 n 3 6 n 3 8 n
12
6 1 n 6 3 n 6 5 n 6 7 n 7 2 n 7 4 n 7 6 n 7 8 n 8 1 n 8 3 n 8 5 n 8 7 n
1
1 2 n
1
2 3 n
1
5 4 k
2
4 3 n 6 3 n

Sample Output

Player 1:
No pieces can be captured.
Player 2:
No pieces can be captured.

Player 1:
(1, 2) to (3, 4), captures (2, 3).
Player 2:
No pieces can be captured.

Player 1:
(5, 4) to (3, 2), captures (4, 3).
(5, 4) to (7, 2), captures (6, 3).
Player 2:
(6, 3) to (4, 5), captures (5, 4).

References:
Daydreams of a Mathematician

When we were kids, we all had those moments where our minds drifted somewhere else and our imagination ran wild. You may have dreamt of fighting dragons, flying in space, being a King or Queen, or perhaps being a firefighter. You may have even had an imaginary friend. Hopefully you still have retained some of your ability to imagine. After all, imagination makes up the ground for those things that are so fantastic that we could never have them – things like, as I am sure you have always longed for, the square root of a little number we call $-1$.

Ah, $-1$, why do you trouble us so? A number merely one less than zero, $-1$ and its friends can pose some pesky problems in mathematics. Mathematicians knew long ago that no amount of prayer or well-wishing would ever bring about the value of $\sqrt{-1}$. So, they reached into that same imagination we all relied on throughout our childhood and created their own number: $i$. Now, the insecurities and the fears of humble mathematicians were solved. In today's world of war, fear, and desperation, at least we all have the peace of mind in knowing that $\sqrt{-1} = i$.

So let's have some fun with our good friend $i$. Turns out that the discovery of $i$ lead to the creation of a whole new set of numbers called the complex numbers. Complex numbers include all real numbers, like 3, $\sqrt{2}$, $1\times10^{100}$ (a googol), and $1\times10^{999}$ (a googolplex). But they also include numbers with an imaginary part, like $2+3i$, $6i$, and $2-i$. All complex numbers can be written in the form $a + bi$ (we'll call this proper form), where $a$ is the real part and $b$ is the imaginary part. Thus, in $2-i$, 2 is the real part and $-1$ is the imaginary part.

Your task is to carry out some basic complex number multiplication. Multiplication in the complex numbers is carried out much like it is in the real numbers, except that you must remember that $i^2 = -1$. Thus, $3i \times -2i = -6i^2 = (-6) \times (-1) = 6$. When multiplying complex numbers of the form $(a+bi)(c+di)$, carry out the multiplication just as you would in the real numbers if you were faced with an equation $(m+n)(x+y) = mx + nx + my + ny$.

For the purpose of this problem, we will define the proper form of a complex number. A complex number in proper form will always have at most one real part and one imaginary part. In proper form, a complex number must be expressed as $a + bi$, if the imaginary part ($b$) is positive or $a - (-b)i$ if $b$ is negative. Furthermore, if either $a$ or $b$ is 0, it should be omitted from the expression. Finally, if $b=1$ or $-1$, it should not appear as a coefficient next to $i$. Instead, $i$ should appear on its own with the proper sign. The following complex numbers are in proper form: $1 - 3i$, $1 - i$, $2$, $i$, and $4 + 5i$. The following complex numbers are not in proper form: $0 + 4i$, $1 + 1i$, $3 + -4i$, and $2 - 0i$. 
Details of the input
There will be multiple test cases. Input will start with an integer k indicating the total number of test cases. Following this line will be k pairs of input lines of the form

\[ \text{complex1} \]
\[ \text{complex2} \]

where complex1 \text{ and complex2} are complex numbers that are not equal to 0. All input numbers will be written in proper form except that the imaginary part may be expressed before the real part (i.e., bi + a as opposed to a + bi). Both the imaginary part and the real part will be integers.

Details of the output
There will be one line of output per test case. Each line of output should be of the form:

Product \( k \) is: \( \text{answer} \).

Where \( k \) is the number of the test case starting with 1 and where \( \text{answer} \) is the product of the complex number multiplication in proper form. To be in proper form, \( \text{answer} \) must appear either as \( a + bi \) if \( b \) is positive, or \( a - bi \) if \( b \) is negative (i.e., \( a + -b \) is incorrect). Furthermore, if either \( a \) or \( b \) is 0, it should be omitted just as it was with the input data. There will be no situations where both \( a \) and \( b \) are 0. Finally, if \( b = 1 \) or \( -1 \), the 1 should not appear as a coefficient next to \( i \) (as was done in the input). Look at the test data carefully.

Sample Input
2
4i - 3
2 + 3i
3 + 2i
-2i + 3

Sample Output
Product 1 is: -18 - i
Product 2 is: 13
Su-do-who?

Just about everyone at this point has been exposed to the craze known as Sudoku. Sudoku is a variation of Latin squares which are a variation of Magic Squares, whose origins traced back to ancient China. The concept of Sudoku is relatively simple. In the most common version of the game, given a 9x9 grid, each cell in a solution is to contain number from 1-9 such that no row or column contains the same number twice. In addition, when subdivided into nine 3x3 grids as indicated in the picture above, no sub-grid contains a duplicate number. Variations of the game will use different sized grids, but with the same basic set of rules.

Sudoku puzzles start with a sparsely populated grid that can only be filled in exactly one way to obtain a legal solution. The number and arrangement of initial values given determine the complexity of finding the solution. Algorithms to solve Sudoku puzzles have been developed, but they are beyond the scope that this problem is intended to cover. Your job will be to write a Sudoku solution validator – that is, a program that given a proposed solution as input, will determine whether or not the solution is, in fact valid.

In order to validate a solution, the grid must be analyzed in three ways – row-by-row, column-by-column, and subgrid-by-subgrid. For each analysis, two types of errors can occur – one or more values in the range 1..n could be missing and/or duplicates of one or more values in the range 1..n could appear.

Details of the input
Input to your program will be as follows. The first line of input will indicate how many instances of Sudoku solutions will follow. Each instance will begin with a line containing 3 integer values indicating the overall size of the grid and the dimensions of each subgrid, rows first, then columns. For the picture above, this line would read 9 3 3

Another example might be 6 2 3 which would define a grid that looks as follows:

```
6 8 2 1 9 4 3 5 7
7 3 1 5 6 8 9 2 4
4 9 6 7 2 3 8 6 1
8 2 7 9 3 5 1 4 6
5 1 9 6 4 7 2 8 3
3 6 4 2 8 1 5 7 9
9 5 6 4 1 2 7 3 8
2 4 8 3 7 9 6 1 5
1 7 3 8 5 6 4 9 2
```

Following this input, will be a series of n lines, each containing n values in the range 1..n, representing the values in the proposed solution reading from top to bottom and left to right in the grid.
**Details of the output**

If a proposed solution is correct, a message to that effect will be printed (see example). If a solution is incorrect, error messages will be generated to indicate what type of error occurred. Error messages related to rows, should precede those for columns, which should precede those for subgrids. In each case, error messages about missing values should precede those about duplicate values. Error messages will take one of the following two forms:

- Row x contains no y.
- Row x contains y values of z.

For error reporting purposes, rows and column are numbered from 1 to n, and the sub-grids of the grid will be numbered from 1 to n with 1 being the upper left subgrid and n being the lower right subgrid. Note that an incorrect solution may have more than one problem and all problems identified should be reported.

Output for each instance will begin with a single line stating the problem instance (see output below) and end with a blank line to separate it from the one following. The final instance also has the blank line following it.

**Sample input**

```
2
4 2 2
1 3 2 4
2 4 3 1
4 2 1 3
3 1 4 2
8 4 2
1 2 3 4 5 6 7 8
2 8 6 4 1 3 5 7
3 5 7 1 6 8 4 4
4 6 8 2 3 7 1 5
5 7 1 3 2 4 8 6
6 1 2 7 8 5 4 3
7 3 5 6 4 1 8 2
8 3 4 5 7 2 6 1
```

**Sample output**

Instance 1:
Instance 1 is correct.

Instance 2:
Row 3 contains no 2.
Row 3 contains 2 values of 4.
Column 2 contains no 4.
Column 2 contains 2 values of 3.
Column 4 contains no 8.
Column 4 contains 2 values of 4.
Column 7 contains no 2.
Column 7 contains no 3.
Column 7 contains 2 values of 4.
Column 7 contains 2 values of 8.
Subgrid 1 contains no 7.
Subgrid 1 contains 2 values of 2.
Subgrid 2 contains no 5.
Subgrid 2 contains 2 values of 4.
Subgrid 3 contains no 2.
Subgrid 3 contains no 4.
Subgrid 3 contains 2 values of 3.
Subgrid 3 contains 2 values of 6.
Subgrid 4 contains no 2.
Subgrid 4 contains no 3.
Subgrid 4 contains no 6.
Subgrid 4 contains 2 values of 4.
Subgrid 4 contains 2 values of 5.
Subgrid 4 contains 2 values of 7.
Subgrid 5 contains no 2.
Subgrid 5 contains no 4.
Subgrid 5 contains 2 values of 3.
Subgrid 5 contains 2 values of 7.
Subgrid 6 contains no 8.
Subgrid 6 contains 2 values of 5.
Subgrid 7 contains no 3.
Subgrid 7 contains no 6.
Subgrid 7 contains 2 values of 2.
Subgrid 7 contains 2 values of 4.
Subgrid 8 contains no 5.
Subgrid 8 contains no 7.
Subgrid 8 contains 2 values of 6.
Subgrid 8 contains 2 values of 8.
Crate Creations

One of the leading dorm furnishing companies in the U.S., Crate Creations, has hired you into the position of Raw-Prep Supervisor. In your department, you are able to handle up to one thousand orders per work week. It is expected that you will organize the department’s materials and order requests such that the products in the orders are pre-cut and ready to assemble for the Assembly Department.

In order to minimize costs, all of the material is cut from two-by-four pieces of wood that are at most twenty-five feet in length. The actual length and cost of each piece of wood depends upon the height of the trees the mill receives each morning. The wood is cut into pieces that are measured in whole feet according to the specifications for each item in the order. Your paperwork (also known as the input to your program) will tell you how many cuts of a particular size will be needed per product. It is your job to group the products of an order together and determine the amount of scrap material that will be left over and the cost of making the order.

To prepare an order, you will begin by taking a full piece of wood and making the longest cut possible from the pieces that you need. Then, with the remaining portion of the board, you will again cut the longest piece you can until you have either used the entire piece of raw material or it is smaller than any piece that is still needed for the order. If you have material left at this point, you will toss the remaining piece of raw material into the scrap bin, keeping track of the total scrap for the order. You can then get a new piece of wood and continue this process until all cuts for the order are complete. For your daily reports, you will need to output the order number, the total length of scrap material, and the total cost of material to complete the order, which depends on the total number of full boards required for the job.

Details of the input
The first line of your paperwork will contain two integers, \( l \) and \( c \), which represent the length and cost (in US dollars) of one full piece of the raw material that you will be using for the day. As stated earlier, \( l \) will be an integer less than or equal to 25. The next several lines will define the material requirements for each of the company’s products. Each line will be of the following form:

\[ \text{name } i \text{n}_1@x_1 \text{n}_2@x_2 \ldots \text{n}_i@x_i \]

where \( \text{name} \) is a product name and \( i \) is the number of entries of the form \( n@x \) that follow on the line. For each \( n@x \), \( n \) represents the number of pieces of length \( x \) needed for the product. For example, in the sample input below a table requires 12 cuts of length 5 and 4 cuts of length 4. There will be no more than ten types of cuts needed for a single product and they will all appear on the same line. The company has at most 10 types of products it produces on any one day. The list of products will be followed by a line containing a single ‘#’.

The lines following the product definitions will pertain to the order requests. The first line of an order will specify the order number in the form ‘Order: xxxx’ where xxxx is in the range ‘0000’–‘9999’. The lines following an order number will define the quantity and name of the particular products needed to complete an order. In the sample input below Order 0001 specifies 2 chairs and 3 tables are to be made. There will be no more than 200 orders per day with the final order being followed by a line containing a single ‘#’.
Details of the output
The output of your program summarizes the amount of scrap and cost of each order and provides a total amount of scrap and cost at the end of the report.

The output for an order is to be of the form

```
Order #xxxx:          (i.e., the order number is preceded by '#' and followed by ':')
Scrap Material: s feet.  (i.e., the cost is preceded by '$' and followed by '.')
Total Cost: $c.
```

and each order's report is separated from the next by a blank line.

The final output should give the totals for all orders in the form

```
Total Scrap: S feet.
Total Cost: $C.
```

Sample Input
10 40
table 2 1@5 4@4
table 2 1@6 1@2 5@1
desk 2 1@4 9@3
#
Order: 0001
2 chair
3 table
Order: 0002
4 chair
1 table
#

Sample Output
Order #0001:
Scrap Material: 4 feet.
Total Cost: $1080.

Order #0002:
Scrap Material: 8 feet.
Total Cost: $640.

Total Scrap: 12 feet.
Total Cost: $1720.