CCSC-E 2014 Programming Contest

November 15th, 2014

PROBLEM #1

Root Beer Collection

As a world traveler who loves (root) beer, you've collected several bottles from various places around the world. Now that you're home, you want to display the bottles and make them look nice. Most people would just line them up on a shelf, but not you. You can't use a line- how could you ever decide which is "first" and which is "last"? Instead, you'll arrange them in a circle. Additionally, since the bottles have different sizes and colors, you've decided on the following display properties:

1. Each pair of adjacent bottles must have the same size or color
2. No pair of adjacent bottles may have both the same size and color.
3. Since you're forming a circle, every bottle is adjacent to two bottles (there is not "start" or "end")
4. All bottles must be used.

Given your set of bottles, can you arrange them in a way that follows the rules?

Input
There will be several test cases. Each test case will start with a number *T*, indicating the number of types of bottles in the set. *T* will be at most 10, and a value of *T*=0 will denote the end of the input.

If *t* is greater than zero, there will then follow *T* lines. Each line will have three numbers separated by spaces:

C S Q

C is an integer ID for the color, *S* is an integer for the size of the bottle (in millimeters), and *Q* is the quantity of the bottle (how many of them we have). All values will be between 1 and 20, inclusive, and there will be at most 10 bottles total.

Sample input:

```
3
1 3 1
2 3 2
2 3 1
4
1 4 1
2 4 1
2 2 1
1 2 1
0
```

Output:

For each test case *c*, output a line:

```
Case C: We can arrange these bottles.
```

or

```
Case C: We cannot arrange these bottles.
```

The test case numbering starts at 1.

Sample output (corresponding to the sample input):

```
Case 1: We cannot arrange these bottles.
Case 2: We can arrange these bottles.
```

PROBLEM #2

Calculator

In a rush to get the operating system out the door, your favorite company has shipped a defective calculator. And since your own company's metrics provide an incentive for code that works more than it rewards code that is important, you decide to write your own simple calculator for the company instead of simply recommending your favorite online calculator. You won't have time to make it fully functional, but it will still qualify for the incentives. Your calculator will just perform simple calculations on two operands. It will work for six operands:

1. Addition: +
2. Subtraction: -
3. Multiplication: *
4. Division: /
5. Mod: %
6. Exponentiation: ^

Input
There will be several lines of input. Each line will contain two operands with an operator between them. There will be spaces between the operands and the operator. The last line will contain the number 0 by itself.

Sample input:

```
1 + 2
3 * 30
2 ^ 4
221 % 10
0
```

Output:
For each line, output the result of the arithmetic computation on a separate line.

Sample output:

```
3
90
16
1
```

PROBLEM #3

Conspiracy Theory

You have an uncle who believes in an elaborate conspiracy theory. He spends much of his time looking for patterns in newspapers and tracking the locations of certain famous people. He claims that one way that people involved in the conspiracy secretly communicate with each other is by using a convoluted code involving the number of letters in the words in a newspaper article.

"Count the number of letters in each word to turn the word into a number. Count the number of times in a line you have a number that divides evenly into the number that follows it. However many times that happens – that's the number for the line."

"What do you do with those numbers?"

"I haven't figured that part out yet. First we need to easily convert a line of text into numbers."

Input
There will be one line of input. Ignore punctuation and numbers. Count only letters.

Sample input #1:

```
This is the sample input.  There is no conspiracy.
```

Sample input #2:

```
This is another sample input.  There is a conspiracy.
```

Output:
Convert the words into numbers based on the number of letters in each word. Words will contain at least one letter and are separated by one or more spaces. Based on the conversion, determine how many numbers in the line are evenly divisible by the previous number in the line. Output the result.

Output for sample input #1:

```
4
```

Output for sample input #2:

```
2
```

Problem #4

The laziest man in the world

Languid von Slothful is a very lazy man. He doesn't like to walk, he doesn't even like to get up from his chair, if he can help it. The problem is that he needs to get to an important meeting across town, and needs to figure out the way to do it while expending the least amount of energy. The town has a series of bus routes, with stops on several corners. However, not all buses go where he needs to go, so Languid needs to figure out a plan. Since he's too lazy to do it himself, he's asked you to help him out.

The town is an MxN grid, which can be traversed rectilinearly (so no diagonals). Each horizontal or vertical street between locations on the grid can be traversed:

- By walking (always)
- By taking one of several bus lines (if the route of the bus includes the street)

Languid's priorities, in order, are:

1) Walk as few blocks as possible.
2) Out of all possible paths in which Languid walks the fewest blocks, choose the one that transfers between the fewest different bus lines (he'd much rather take a long indirect path than have to get up and move to a different bus).

If there are multiple paths that involve the fewest blocks in which Languid walks and the fewest bus transfers, any possible path is fine.

Input

There will be several input instances. Each input will begin with 2 non-negative integers *m* and *n*, which define the number of rows and columns in the grid. (m and n are both <= 100) Values of m and n = 0 denote the end of the input. Otherwise, there will be two pairs of integers, one pair per line, *r* and *c* which denote the starting and ending row and column of the locations Languid must travel between. Input instances will be separated by a single blank line.

Next will come a positive integer *B* (*B* <= 20), denoting the number of bus lines. Then will be *B* bus descriptions. Each description will begin with an integer *K*, for the number of streets connected by the bus. There will follow K pairs, *r* and *c* which mark the grid locations serviced by the bus route.

Each location will be connected to at least one other location one street away, and it will be possible to get between any pair of locations serviced by the bus following the bus route (in other words, it will not be the case that one bus route has multiple disjoint components). All bus lines will be connected to each other (though possibly not directly), so there will be a path using some number of bus lines between any two locations that have a bus stop.

All row values will be between 0 and n-1, and all column values will be between 0 and m-1

Output:
For each case *i*, starting with 1, output:
```
     Case i: Walking a, Riding b
```
where *a* is the number of blocks needed to walk, and *b* is the number of different bus lines used. There is a newline after each case, including the last one.

Sample Input:
```
     3 6
     1 0
     1 5
     2
     3
     1 1
     1 2
     1 3
     2
     1 3
     1 4

     3 6
     1 0
     1 5
     3
     2
     1 1
     1 2
     2
     1 3
     1 4
     3
     1 2
     1 3
     2 3
     0 0
```

Sample Output:
```
Case 1: Walking 2, Riding 2
Case 2: Walking 2, Riding 3
```

PROBLEM #5

Watching the Clock

In TV and movies, you often see shorthand for time passing as a picture of a clock spinning round and around. In this problem, you're asked to see what time the clock says once it stops spinning.

Input:

There will be several input instances. Each instance will have two numbers: *h* (between 1 and 12 inclusive), representing a clock time in hours, and *r* (between -1000 and 1000, inclusive), representing a number of rotations to the clock. The value of *h* will be an integer. The value of *r* may or may not be an integer. A positive number means you're turning the clock that many hours ahead, a negative number means you're turning the clock that many hours backwards.

A value of *h* = *r* = 0 will denote end of input.

Output:

For each input instance *i*, starting with 1, output the line:

```
Case i: h:mm
```

In this line, *"h"* is where the hours hand of the clock will be pointing after c rotations of the clock, and *"mm"* is where the minutes hand of the clock will be pointing. After computing the time, round to the nearest minute.

Sample Input:

```
1 9.2
11 4.5
1 -4
11 -4.5
1 0.0000001
1 0.0084
0 0
```

Sample Output:

```
Case 1: 10:12
Case 2: 3:30
Case 3: 9:00
Case 4: 6:30
Case 5: 1:00
Case 6: 1:01
```