Eighth Annual Ohio Wesleyan University Programming Contest
March 22, 2014

Rules:

1. There are six questions to be completed in four hours.
2. All questions require you to read the test data from standard input and write results to standard output. You cannot use files for input or output. **Do not** include any prompts, other debugging information, or any other output except for exactly what is specified by the problem.
3. The allowed programming languages are C, C++ and Java.
4. All programs will be re-compiled prior to testing with the judges' data.
5. Non-standard libraries cannot be used in your solutions. The Standard Template Library (STL) and C++ string libraries are allowed.
6. Programming style is not considered in this contest. You are free to code in whatever style you prefer. Documentation is not required.
7. All output cases will end in a newline character. So after your program finishes output, the cursor should be on the first column of the line immediately below your last case's output.
8. All communication with the judges will be handled in the PC2 environment.
9. Judges' decisions are final. No cheating will be tolerated.

Problem A: Keeping an Eye on the Kids

The old lady who lived in a shoe didn't spend all of her time there. Sometimes, she gathered up all of her children and took them to parks, stores, or museums. The problem is that when she takes her kids to these places, they go off in so many directions she doesn't know what to do, especially when they hide behind walls, or in different rooms. She'd like to find a place to stand that gives her a view of as many children as possible. Can you help her evaluate possible standing positions?

Input:

Each case will begin with an integer *n* (≤ 50), denoting the number of obstacles. A value of *n* =0 will denote end of input. Otherwise, there will then follow *n* lines with 4 integers each on it:

x_s y_s x_f y_f

x_s and y_s form the x and y coordinates of the start of an obstacle. x_f and y_f are the x and y coordinates of the end of the obstacle. All obstacles are straight lines.

The case will continue with an integer *c* (≤50), holding the number of children. Then there will be *c* lines, each line holding the x and y position of a child.

The case will continue with an integer *p* (≤50), holding the number of positions the old lady is considering standing in. There will then be *p* lines of 2 integers holding the x and y coordinate of her position.

All points in one case (whether they are endpoints of obstacles, locations of children, or the location of the lady) will be distinct. It is possible for two obstacles to cross one another. It is not possible for a child, or one of the lady's positions to coexist with an obstacle.

Output:

For each input case *i* and position *j* (both starting from 1), output the line:

`Case i, Position j: Children seen: k`

Where *k* is the number of children the old lady can see. She can see a child if there is a straight line from her to the child that does not intersect any of the obstacles. None of the lines from the lady to any child will ever hit an endpoint of an obstacle (they will either intersect somewhere along the line, or will not intersect at all)

Output a blank line after all of the positions in a case including the last one

Sample Input:

```
1
6 5 10 9
3
2 3
8 1
8 8
2
10 5
3 4
3
8 9 1 9
0 8 6 13
7 8 7 14
5
4 1
2 13
9 12
5 11
4 10
3
4 6
6 10
8 11
0
```

Sample Output:

```
Case 1, Position 1: Children seen: 2
Case 1, Position 2: Children seen: 3

Case 2, Position 1: Children seen: 1
Case 2, Position 2: Children seen: 2
Case 2, Position 3: Children seen: 1
```

Problem B: N-Puzzle Solvability

A popular class of puzzles is the "N" puzzle, where you are given an NxN grid of tiles with tiles numbered from 1 to N^2-1, with one blank space (where there is no tile). So, if N=3, a possible board would be:

5	3	1
7	2	8
4		6

The goal is to rearrange the board by sliding tiles into the blank space (creating a blank space in a different location) and getting the tiles in numerical order:

1	2	3
4	5	6
7	8	

It turns out that not every puzzle is solvable. Here are the conditions for determining if a particular instance of the puzzle is solvable.

- List out the numbers in the puzzle in one long list, row by row. So the starting puzzle above would create the following list:

5	3	1	7	2	8	4		6

- Count the number of <u>inversions</u> in the list. An inversion is a pair of non-empty indices, i and j, where i < j and List[i] > List[j] (meaning that position i and position j are out of order). The above puzzle has 11 total inversions.

- Apply the following rule:
  - If n is odd, the puzzle is solvable if and only if it has an even number of inversions
  - If n is even, count the number of rows from the bottom the blank is on (starting from 1). If the blank is on an odd row, the puzzle is solvable if and only if there are an even number of inversions. If the blank is on an even row, the puzzle is solvable if and only if there are an odd number of inversions.

Your task is to determine the solvability of a given puzzle.

Input:

There will be several input instances. Each will begin with a number *n*, 2 ≤*n* ≤100. A value of *n*=0 will signify the end of input. Otherwise, there will be *n* lines, each line will have *n* numbers, separated by spaces. The numbers will all be in the range between 0 and n^2-1, and each number will appear exactly once. The 0 will signify the blank space. Note that the 0 is not a tile, and should not be used for counting inversions.

Output:

For each input instance *i*, output the line:

`Case i: K inversions, so yes.`

Or

`Case i: K inversions, so no.`

Where "*K*" is the number of inversions found in the given puzzle instance, and "yes" is output if the puzzle is solvable, and "no" is output otherwise

Sample Input:

```
3
5 3 1
7 2 8
4 0 6
4
2 8 11 9
0 1 12 15
6 7 4 3
5 10 13 14
4
2 8 11 9
0 1 12 15
6 7 4 3
5 10 14 13
0
```

Sample Output:

```
Case 1: 11 inversions, so no.
Case 2: 42 inversions, so yes.
Case 3: 43 inversions, so no.
```

Problem C: Splash Height

Little Joey is taking a bath. He likes to splash in the bathtub and see how high of a splash he can make. Interestingly, all of his splashes always go an integral number of inches above the water. Joey has time for N splashes before bathtime is over. Each splash gets a response from the parent watching him, depending on how it relates to the maximum legal splash height H that the parent will accept.

- If the splash height was < H, there is no response from the parent (the splash was not too high)
- If the splash height was > H, Joey gets a warning that the splash was too high. Joey has some number K of warnings until the parent runs out of patience. If Joey has used all K warnings and again splashes too high, the bath immediately ends (even if he had time for more splashes) and Joey gets sent straight to bed without TV. This is not a desirable outcome.
- If the splash height was = H, Joey gets told "That's as high as you can go- no higher!" This does not consume one of the K warnings, and Joey's goal is to get this message (because it means he has found the maximum legal splash height)

Joey's goal is to make at least one of his N splashes be of height H, (which seems to be different each night), without splashing over H more than K times, and also without using more than N total "test splashes" (N and K seem to change every night too). Little Joey doesn't know Computer Science (yet), but you do, and you know that given values of N and K, you can develop an algorithm that will use "test splashes" to find H, as long as it's between 1 and some M. Your job is to find this M, given values of K and N.

Note that your job is not to find H. Your job is to figure out what the maximum largest legal splash we can find is, should we care to look for it. In other words, for some values of H, N, and K, it is possible to figure out H in N splashes with K warnings, but for some other values, it is impossible (H is too large relative to N and K, so we can't find H exactly given the limited number of splashes at our disposal). The M you're looking for is the largest H we can guarantee that we'll find given N and K.

For example, if N=2 and K=0, then the only heights we can test are 1 and 2 (in that order, since we can't go over H) , so M=2. If N=2 and K=1, we'd first test 2, and based on what we were told, next splash either 1 or 3 next, so M=3.

Input:

There will be multiple input instances. Each instance will consist of two integers, K and N ($1 \leq K \leq N \leq 30$). A value of K and $N < 0$ will signify end of input.

Output:

For each input instance *i*, output the line:

`Case i: We can find the maximum splash height if it's between 1 and M inches.`

..where "*M*" is the value defined above.

Sample Input:

```
0 5
1 5
1 2
5 5
15 20
-1 -1
```

Sample Output:

```
Case 1: We can find the maximum splash height if it's between 1 and 5 inches.
Case 2: We can find the maximum splash height if it's between 1 and 15 inches.
Case 3: We can find the maximum splash height if it's between 1 and 3 inches.
Case 4: We can find the maximum splash height if it's between 1 and 31 inches.
Case 5: We can find the maximum splash height if it's between 1 and 1047224 inches.
```

Problem D: Orders of Age Magnitude

Last year, I turned 40 while my son turned 4. We'll stay this way until he turns 5 later this year. I've tried to convince my friends how awesome it is that I'm an order of magnitude older than my child, but for some reason, nobody thinks it's as cool as I do. But now, I have a captive audience. Who not only will have to hear me say how cool it is, but will also write a program that will determine when this correspondence will happen for people in general. Won't that be even cooler?

Input:

There will be multiple input instances. Each input will contain two 4-digit integers between 1900 and 2100 (inclusive), *Y1* and *Y2*. (*Y1*≠*Y2*). *Y1* will be the birth year of the person (parent or child) that has the earlier birthdate in the year (so in a given calendar year, the person whose birth year is *Y1* will have their birthday first). A value of *Y1*=*Y2*=0 will signify the end of input.

Output:

For each input instance *i*, output either the line:

`Case i: In Year Y, the parent will be A1 while the child will be A2.`

(Where *A1* is the parent's age when they are 10x as old as the child, *A2* is the child's age when this happens, and *Y* is the first year this happens in)

Or the line:

`Case i: No such correspondence is possible.`

..which you will output if no year exists where *A1* is ≤ 100 (after that, we'll say the parent is too old).

Sample Input:

1973 2009
2009 1919
1974 2009
2009 1974
0 0

Sample Output:

```
Case 1: In Year 2013, the parent will be 40 while the child will be 4.
Case 2: In Year 2019, the parent will be 100 while the child will be
10.
Case 3: In Year 2014, the parent will be 40 while the child will be 4.
Case 4: No such correspondence is possible.
```

Problem E: Getting the Goods in Barterville

As a lowly apprentice for the local potion maker of Barterville, you are often sent down to the Barterville Marketplace to stock up on supplies. The Barterville Marketplace is distinctive in that:

- All merchants only exchange goods for other goods in some combination (there is no money in Barterville)
- Each merchant has one preferred trade, and will do it as many times as you want, in either direction

On a given day, you're sent to the marketplace with a set of goods to begin trading with, and have a different set of goods you need to bring back to the potion maker. You'd like to do this in as few trades as possible, and carry as little extra back as possible. How many trades will you need to make?

Input:

There will be several input instances. Each input instance will begin with an integer *N* ($N \leq 26$) denoting the number of goods available in the town. A value of *N*=0 will denote the end of input. The goods will be named with the first *N* letters of the alphabet

If *N*>0, then there will be a positive integer *M* (≤7), denoting the number of merchants. There will then be *M* strings of the following format:

<list of goods>=<list of goods>

There will be no spaces in the strings. Each good will be a capital letter (one of the first *N* letters of the alphabet).

After the *M* strings for the *M* merchants, there will be two more strings. The first string will be a list of the goods you start out with. The second string will be a list of goods you're required to bring back (though you may be bringing back more).

All strings will be displayed in alphabetical order (for trades, each side of the trade will be in alphabetical order). If a character is repeated in a string, then that good is present (or needs to be present) multiple times. The merchant trades will have a maximum size of 5 characters per side. The starting and goal good lists will have a maximum length of 5 characters.

Output:

For each input instance *i* that has a successful trade sequence of size 10 or less, output the line:

`Case i: Trading complete in j trades, with k extra.`

..where *j* is the number of trades you made to get the goods you need to bring back, and *k* is the number of extra goods (in excess of the required ones) you'll be bringing back. If multiple trade sequences exist

that allow you to reach your goal, break ties first to minimize the number of trades, and second to minimize the number of excess items.

If no sequence of 7 trades or less exist that will allow you to gain the requested goods, output the line:

`Case i: No sequence of 7 trades or less exists.`

Sample Input:

```
4
2
A=BC
C=CD
A
BCD
4
3
A=BB
AA=B
AB=CCDD
B
AABBCCD
5
5
A=B
B=C
C=D
D=E
E=A
A
AA
0
```

Sample Output:

```
Case 1: Trading complete in 2 trades, with 0 extra.
Case 2: Trading complete in 7 trades, with 2 extra.
Case 3: No sequence of 7 or less trades exists.
```

Problem F: Daring Dan vs the Space Amoebas

Daring Dan is a space explorer who finds himself on an alien planet. On this planet there exists a ravenous space amoeba that grows in all directions. Only armed with his trusty blaster, and stuck to the floor, unable to move, Daring Dan needs to hold out for as long as possible in the hopes that help will arrive. How long can he last?

The area Dan is located in can be represented as a two-dimensional grid. Some grid squares are empty, some grid squares contain space amoebas, some grid squares contain impassable barriers, and one square contains Dan himself. It takes Dan one minute to clear a grid square of an amoeba. He can only shoot in the 4 cardinal directions (North, South, East, and West) and will shoot the closest amoeba he can see (he can't see through walls), clearing only the square he shoots. If Dan can't see any amoebas, he will do nothing during this minute. It then takes one minute for Dan's blaster to recharge. During that minute, all amoebas will "expand" into adjacent empty rectilinear grid squares. (In other words, once the recharge minute is done, all empty squares that are adjacent (by North, South, East, and West) to amoeba squares will also contain amoeba squares in addition to the squares the amoeba was in previously). If an amoeba can expand into Dan, it will, eating him.

Even if Dan could not see any amoebas (and so did not shoot any), Dan can't shoot any newly grown amoeba until it is finished expanding. So, Dan can only shoot amoebas in odd numbered minutes (starting at 1, when he gets the first shot) , and amoebas can only grow in even numbered minutes.

Input:

There will be several input instances. Each instance will begin with two integers, r and c (both ≤ 20), denoting the number of rows and columns in the grid. A value of $r=c=0$ will denote end of input.

Otherwise, there will follow r lines of c characters, each character on a row will be separated by a space. Each character will either be:

'A': meaning the corresponding square of the grid holds an amoeba

'X': meaning the corresponding square of the grid bolds an impassable barrier

'-': meaning the corresponding square of the grid is empty

'D': meaning the corresponding square of the grid holds Dan (exactly one square will have this property)

Output:

For each input instance *i*, output either the line:

`Planet i: Dan will be eaten in k minutes!`

…Where *k* is the number of minutes Dan can last before being eaten

Or:

`Planet i: Dan can hold out indefinitely!`

…If Dan can stay safe for any amount of time.

Sample Input:

```
10 10
- - - - A - - - - -
- - - - - - - - - -
- - - - - - - - - -
- - - - - - - - - -
- - - - - - - - - -
A - - - D - - - - A
- - - - - - - - - -
- - - - - - - - - -
- - - - - - - - - -
- - - - A - - - - -
5 6
- - A X X -
- - X - - -
- - D - - -
- - X - - -
- - - - - -
3 4
- - X -
- D A A
- - X -
0 0
```

Sample Output:

```
Planet 1: Dan will be eaten in 12 minutes!
Planet 2: Dan will be eaten in 24 minutes!
Planet 3: Dan can hold out indefinitely!
```