Ninth Annual Ohio Wesleyan University Programming Contest
March 28, 2015

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. <u>**Do not**</u> 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 PC^2 environment.
9. Judges' decisions are final. No cheating will be tolerated.

Problem A: Cancelling Fractions

One of the things they teach you in school is that when you cancel fractions, you don't cancel the digits (instead, you cancel the factors). So, for example you can't take the fraction:

$\frac{12}{24}$, cancel the 2 from the top and bottom, and be left with the fraction $\frac{1}{4}$.

Except, sometimes you can. If you start with the fraction $\frac{16}{64}$, cancel the "6" from the top and bottom, you're left with $\frac{1\cancel{6}}{\cancel{6}4} = \frac{1}{4}$, which are actually equivalent.

Your job is to find out whether a fraction is "number-cancelable" in this way.

Input:

There are several input instances, one on each line. Each instance will be two integers, *n* and *d*, each between 10 and 100,000, separated by a space. A value of *n*=*d*=0 will signify end of input. The number *n* will signify the numerator of a fraction, and the number *d* will be the denominator. Each *n* and *d* pair will have at least one digit in common (but may have more).

Output:

For each input case *i*, output either:

```
Case i: n
```

Where *n* is a digit we can cancel from both the numerator and denominator, producing an equivalent fraction. If multiple digits can be cancelled, choose the smallest.

Or, if no canceling is possible, output:

```
Case i: Not possible
```

Sample Input:

```
16 64
160 640
12 24
0 0
```

Sample Output:

```
Case 1: 6
Case 2: 0
Case 3: Not possible
```

Problem B: Good Guys and Bad Guys

Little Joey is setting up figures on a board. There are two kinds of figures: “Good Guys” and “Bad Guys.” Once the figures are set up, he turns to you and asks you to pair off the figures so every good guy is fighting a different adjacent bad guy (and vice versa). Can you help him?

Input:

There will be multiple input instances. Each input instance will begin with a number *N* (≤100), denoting the number of each kind of piece (so the board will have *N* good guys and *N* bad guys). A value of *N* = 0 will signify end of input. Otherwise, next will follow *N* pairs of numbers *r c*, the row and column of each good guy, followed by *N* more pairs of numbers *r c*, the row and column of each bad guy. All *r* and *c* values will be between 0 and 1000 (inclusive)

Output:

For each input instance *i*, output either:

```
Case i: Possible
```

Or

```
Case i: Not possible
```

A pairing is “possible” if we can pair the figures into groups of one good guy and one bad guy where:

- The two figures are rectilinearly adjacent (no diagonals)
- Each figure is in exactly one pairing

Sample Input:

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

Sample Output:

```
Case 1: Not possible
Case 2: Possible
```

Problem C: Iterated Bubbles

When you blow bubbles, you stick the bubble wand into a dish filled with solution and blow. When you blow, several bubbles come out. It is possible to catch one of the bubbles you've blown on the bubble wand, and re-blow, giving yourself a new set of bubbles. When you do this, you will have less bubbles in the new set, and they will all use less bubble solution. You can repeat this process by catching one of the new bubbles, and so on.

Suppose you're a good enough bubble blower that each time you blow bubbles, you always blow a new amount of bubbles that is a constant percentage of the number of bubbles you blew the last time (rounding down in the case of fractional bubbles), and all of the new bubbles that are re-blown evenly distribute the bubble solution you are currently using. Then, this process of re-blowing bubbles will stop in one of two ways:

- Either the number of bubbles you are about to blow gets rounded down to zero bubbles, or:
- Each bubble that you are to blow is not given enough solution to form the outline of a bubble (so the bubbles never get created)

How many sets of bubbles can you blow from the initial amount of bubble solution?

Input:

There will be several input instances. Each input instance will begin with an integer *s*. A value of s = 0 denotes end of input. Otherwise, *s* is the amount of solution you start with, and three more integers, *n*, *p*, and *m*, will follow. *n* is the number of bubbles initially blown, *p* is the percentage of bubbles you want to retain when you re-blow, and *m* is the minimum amount of solution you need to successfully blow bubbles. You can assume $s \geq m$, and that the initial solution is evenly distributed among the *n* initial bubbles. The maximum value of *p* is 100, the maximum values of *s*, *n*, and *m* are 1,000,000,000.

Output:

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

```
Case i: b
```

Where *b* is the number of times you can blow bubbles using the method above.

Sample Input:

```
100000 300 50 1
10000 10 2 1
500000 20 90 1
0 0 0 0
```

Sample Output:

```
Case 1: 2
Case 2: 1
Case 3: 4
```

Problem D: Count the Kingdoms

Minnesota Smith, the famed archaeologist, has come to you with a problem. She has been excavating an area of the wilderness that, in the past, was home to some unknown number of kingdoms. She has found a collection of tombstones stating the starting and ending dates of each king's reign, but not the countries the king was in charge of. (Apparently the people who made the tombstones thought it was obvious. They were wrong). What's worse, she has only found the tombstones for some of the kings- the rest have been lost. So it's not possible to get a complete chronology of the rulers.

However, realizing each kingdom can only be ruled by one king at a time, Minnesota has asked you to determine a lower bound on the number of kingdoms.

For example, if you had three tombstones: King Albert, who reigned from year 1 to year 10, King Bernard, who reigned from year 2 to year 20, and King Charles who reigned from year 12 to year 30, you can say that there are at least two kingdoms (Albert and Charles could possibly be from the same kingdom). If instead, Charles reigned from years 9-30, then they would all have to rule different kingdoms.

Input:

There will be several input instances. Each instance will begin with an integer *n* (≤100), denoting the number of tombstones found. A value of *n*=0 will denote end of input. Otherwise, there will n pairs of integers *s* and *e* (each ≤ 1000000, *s* ≤ *e*) marking the start and end years of the king's reign. Since you only have information about the years, you should treat two kings who reign in the same year as overlapping (and thus need to have led different kingdoms)

Output:

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

```
Case i: k
```

Where k is the maximal lower bound on the number of kingdoms you have found.

Sample Input:

```
3
1 10
2 20
12 30
3
1 10
2 20
9 30
0
```

Sample Output:

```
Case 1: 2
Case 2: 3
```

Problem E: Hunting for Slimes

In a certain mine-based crafting game, the three-dimensional world is divided into discrete "blocks" of space. A cube of blocks of some size NxNxN is called a "chunk" (the value of N is constant for each instance of the world). The game system has pre-designated certain chunks as ones that spawn a special monster, called a slime, which you want to hunt down. You've managed to excavate a three-dimensional rectangular area, and you're wondering whether any slimes will show up. Since you do know the size of the game's chunks, but you don't know where the boundaries between chunks are, it's possible that your excavated region spans multiple chunks. The question is: What is the maximum and minimum number of chunks your region can intersect with?

Input:

There will be several input instances. Each input instance will consist of four numbers: *n x y z*, all ≤ 100. A value of 0 for all values will mean end of input. Otherwise, *n* is the size of the chunk (in cells), and *x y* and *z* are the dimensions (in cells) of the region you have excavated.

Output:

For each input instance *i*, output the line:
Case *i*: Min *a*, Max *b*

Where *a* is the smallest number of chunks your region could intersect, and *b* is the largest. There is a single space after the comma.

Sample Input:

```
9 11 11 11
1 5 5 5
20 8 9 10
0 0 0 0
```

Sample Output:

```
Case 1: Min 8, Max 27
Case 2: Min 125, Max 125
Case 3: Min 1, Max 8
```

Problem F: Prefixes of Suffixes

A prefix of a string is a substring that begins with the first character. A suffix of a string is a substring that ends with the last character. Given two strings, it's pretty easy to find the longest common prefix. For example, the longest common prefix of "contest" and "convict" is "con". Some string matching algorithms want to find the longest common prefix of all suffixes of a string. This isn't very interesting with a string like "contest", but if I have a string like "abacab", we get the following table

Original String	Suffix	Longest common prefix length
abacab	abacab	6
abacab	bacab	0
abacab	acab	1
abacab	cab	0
abacab	ab	2
abacab	b	0

Rather than making you output an entire list of numbers, we'll gather the results into one number by adding them all together. So in this case, you'd output 9.

Input:

There will be several input instances, each consisting of one string per line. Each string will consist entirely of lower-case letters, and be of maximum length 100. If the string is exactly the word "end," then that signals end of input. Otherwise, the string is to be processed using the rules given above.

Output:

For each input string *i*, output:

```
Case i: n
```

Where *n* is the sum of the length of the longest common prefix of the string with all of its suffixes.

Sample Input:

```
abacab
abacabacab
end
```

Sample Output:

```
Case 1: 9
Case 2: 20
```