Final Assignment

Each group is to submit one solution to eConestoga by Friday, December 11 before class.

Your group will be expected to present your solution to the class during the final class on December 11.

Your mark on this assignment will depend on a combination of the quality, functionality, and adhesion to coding standards of your code. The novelty and/or elegance of your solution will also be taken into consideration.

If you are absent without excuse from this class will result in a mark of zero for the presentation portion of the mark (20%)

I may ask anyone in the group to explain any portion of the code. The clarity of your response WILL affect your personal grade on this assignment.

Group 1 Problem - Time and motion

Movement has long been used to measure time. For example, the ball clock is a simple device which keeps track of the passing minutes by moving ball-bearings. Each minute, a rotating arm removes a ball bearing from the queue at the bottom, raises it to the top of the clock and deposits it on a track leading to indicators displaying minutes, five-minutes and hours. These indicators display the time between 1:00 and 12:59, but without 'a.m.' or 'p.m.' indicators. Thus 2 balls in the minute indicator, 6 balls in the five-minute indicator and 5 balls in the hour indicator displays the time 5:32.

Unfortunately, most commercially available ball clocks do not incorporate a date indication, although this would be simple to do with the addition of further carry and indicator tracks. However, all is not lost! As the balls migrate through the mechanism of the clock, they change their relative ordering in a predictable way. Careful study of these orderings will therefore yield the time elapsed since the clock had some specific ordering. The length of time which can be measured is limited because the orderings of the balls eventually begin to repeat. Your program must compute the time before repetition, which varies according to the total number of balls present.

Operation of the Ball Clock

Every minute, the least recently used ball is removed from the queue of balls at the bottom of the clock, elevated, then deposited on the minute indicator track, which is able to hold four balls. When a fifth ball rolls on to the minute indicator track, its weight causes the track to tilt. The four balls already on the track run back down to join the queue of balls waiting at the bottom in reverse order of their original addition to the minutes track. The fifth ball, which caused the tilt, rolls on down to the five-minute indicator track. This track holds eleven balls. The twelfth ball carried over from the minutes causes the five-minute track to tilt, returning the eleven balls to the queue, again in reverse order of their addition. The twelfth ball rolls down to the hour indicator. The hour indicator also holds eleven balls, but has one extra fixed ball which is always present so that counting the balls in the hour indicator will yield an hour in the range one to twelve. The twelfth ball carried over from the five-minute indicator causes the hour indicator to tilt, returning the eleven free balls to the queue, in reverse order, before the twelfth ball itself also returns to the queue.

Input

The input defines a succession of ball clocks. Each clock operates as described above. The clocks differ only in the number of balls present in the queue at one o'clock when all the clocks start. This number is given for each clock, one per line and does not include the fixed ball on the hour indicator. Valid numbers are in the range 27 to 127. A zero signifies the end of input. Output For each clock described in the input, your program should report the number of balls given in the input and the number of days (24-hour periods) which elapse before the clock returns to its initial ordering.

Sample Input

```
30
45
0
```

Output for the Sample Input

```
30 balls cycle after 15 days.
45 balls cycle after 378 days.
```

Group 2 Problem - System Dependencies

Components of computer systems often have dependencies--other components that must be installed before they will function properly. These dependencies are frequently shared by multiple components. For example, both the TELNET client program and the FTP client program require that the TCP/IP networking software be installed before they can operate. If you install TCP/IP and the TELNET client program, and later decide to add the FTP client program, you do not need to reinstall TCP/IP.

For some components it would not be a problem if the components on which they depended were reinstalled; it would just waste some resources. But for others, like TCP/IP, some component configuration may be destroyed if the component was reinstalled.

It is useful to be able to remove components that are no longer needed. When this is done, components that only support the removed component may also be removed, freeing up disk space, memory, and other resources. But a supporting component, not explicitly installed, may be removed only if all components which depend on it are also removed. For example, removing the FTP client program and TCP/IP would mean the TELNET client program, which was not removed, would no longer operate. Likewise, removing TCP/IP by itself would cause the failure of both the TELNET and the FTP client programs. Also if we installed TCP/IP to support our own development, then installed the TELNET client (which depends on TCP/IP) and then still later removed the TELNET client, we would not want TCP/IP to be removed.

We want a program to automate the process of adding and removing components. To do this we will maintain a record of installed components and component dependencies. A component can be installed explicitly in response to a command (unless it is already installed), or implicitly if it is needed for some other component being installed. Likewise, a component, not explicitly installed, can be explicitly removed in response to a command (if it is not needed to support other components) or implicitly removed if it is no longer needed to support another component.

Input

The input will contain a sequence of commands (as described below), each on a separate line containing no more than eighty characters. Item names are case sensitive, and each is no longer than ten characters. The command names (**DEPEND**, **INSTALL**, **REMOVE** and **LIST**) always appear in uppercase starting in column one, and item names are separated from the command name and each other by one or more spaces. All appropriate **DEPEND** commands will appear before the occurrence of any **INSTALL** dependencies. The end of the input is marked by a line containing only the word **END**.

Command Syntax	*Interpretation/Response*
`DEPEND item1 item2 [item3 ...]`	**item1** depends on **item2** (and **item3** ...)
`INSTALL item1`	install **item1** and those on which it depends
`REMOVE item1`	remove **item1**, and those on whch it depends, if possible
`LIST`	list the names of all currently-installed components

Output

Echo each line of input. Follow each echoed **INSTALL** or **REMOVE** line with the actions taken in response, making certain that the actions are given in the proper order. Also identify exceptional conditions (see *Expected Output*, below, for examples of all cases). For the **LIST** command, display the names of the currently installed components. No output, except the echo, is produced for a **DEPEND** command or the line containing **END**. There will be at most one dependency list per item.

Sample Input

```
DEPEND   TELNET TCPIP NETCARD
DEPEND TCPIP NETCARD
DEPEND DNS TCPIP NETCARD
DEPEND  BROWSER   TCPIP  HTML
INSTALL NETCARD
INSTALL TELNET
INSTALL foo
REMOVE NETCARD
INSTALL BROWSER
INSTALL DNS
LIST
REMOVE TELNET
REMOVE NETCARD
REMOVE DNS
REMOVE NETCARD
INSTALL NETCARD
REMOVE TCPIP
REMOVE BROWSER
REMOVE TCPIP
LIST
END
```

Output for the Sample Output

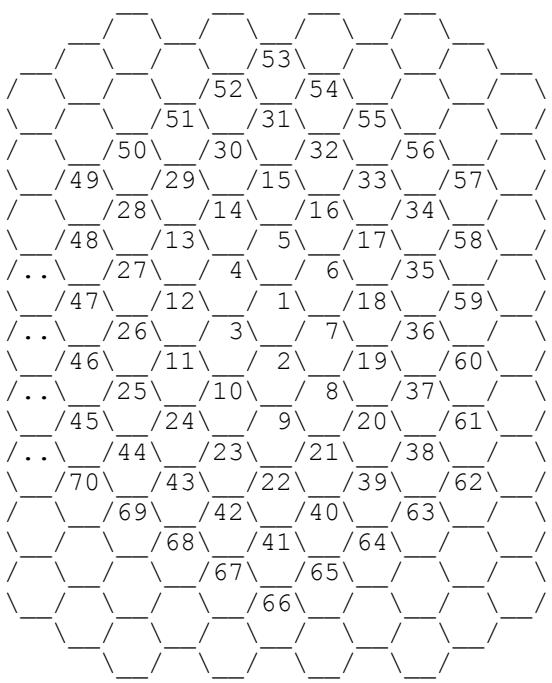
```
DEPEND   TELNET TCPIP NETCARD
DEPEND TCPIP NETCARD
DEPEND DNS TCPIP NETCARD
DEPEND  BROWSER   TCPIP  HTML
INSTALL NETCARD
   Installing NETCARD
INSTALL TELNET
   Installing TCPIP
```

```
   Installing TELNET
INSTALL foo
   Installing foo
REMOVE NETCARD
   NETCARD is still needed.
INSTALL BROWSER
   Installing HTML
   Installing BROWSER
INSTALL DNS
   Installing DNS
LIST
   HTML
   BROWSER
   DNS
   NETCARD
   foo
   TCPIP
   TELNET
REMOVE TELNET
   Removing TELNET
REMOVE NETCARD
   NETCARD is still needed.
REMOVE DNS
   Removing DNS
REMOVE NETCARD
   NETCARD is still needed.
INSTALL NETCARD
   NETCARD is already installed.
REMOVE TCPIP
   TCPIP is still needed.
REMOVE BROWSER
   Removing BROWSER
   Removing HTML
   Removing TCPIP
REMOVE TCPIP
   TCPIP is not installed.
LIST
   NETCARD
   foo
END
```

Group 3 Problem - Bee Breeding

Professor B. Heif is conducting experiments with a species of South American bees that he found during an expedition to the Brazilian rain forest. The honey produced by these bees is of superior quality compared to the honey from European and North American honey bees. Unfortunately, the bees do not breed well in captivity. Professor Heif thinks the reason is that the placement of the different maggots (for workers, queens, etc.) within the honeycomb depends on environmental conditions, which are different in his laboratory and the rain forest.

As a first step to validate his theory, Professor Heif wants to quantify the difference in maggot placement. For this he measures the distance between the cells of the comb into which the maggots are placed. To this end, the professor has labeled the cells by marking an arbitrary cell as number 1, and then labeling the remaining cells in a clockwise fashion, as shown in the following figure.



For example, two maggots in cells 19 and 30 are 5 cells apart. One of the shortest paths connecting the two cells is via the cells 19 - 7 - 6 - 5 - 15 - 30, so you must move five times to adjacent cells to get from 19 to 30.

Professor Heif needs your help to write a program that computes the distance, defined as the number of cells in a shortest path, between any pair of cells.

Input

The input consists of several lines, each containing two integers *a* and *b* (*a*,*b* <= 10000), denoting numbers of cells. The integers are always positive, except in the last line where *a*=*b*=0 holds. This last line terminates the input and should not be processed.

Output

For each pair of numbers (*a*,*b*) in the input file, output the distance between the cells labeled *a* and *b*. The distance is the minimum number of moves to get from *a* to *b*.

Sample Input

```
19 30
0 0
```

Output for the Sample Input

```
The distance between cells 19 and 30 is 5.
```

Group 4 Problem – Scratch and Win

Play the "MOE Millions" scratch and win card and you could win up to $1,000,000! Or you could win less than that. Or you might win nothing at all. There are 10 possible prizes: $1, $2, $5, $10, $50, $100, $1,000, $10,000, $500,000 and $1,000,000.

To play, scratch off the squares on a 3x3 grid one at a time. If you find 3 matching prize amounts under the scratchy stuff, you win that amount. It's that simple! Each card will contain a maximum of one set of 3 matching symbols.

You have a card and you have scratched off 8 of the squares. What fabulous prize could you win when you scratch off that final square?

Input

A test case will consist of nine lines representing the nine squares on the card. The first line is for the top left box, the second is for the top middle box, then top right, then middle row left, and so on down to the bottom right corner. If a box has been scratched, the line for that box will contain the prize amount that is revealed. If not, the line will contain a question mark. The cards could be in any state of play, ranging from just starting out (no boxes scratched yet) to completely finished (all boxes scratched).

Output

Your job is to output a list of all prizes the cardholder can or will win in order from lowest to highest, separated by spaces. Each card in the input should be represented by a single line in your output. If no prize is possible, output the exact string "No Prizes Possible".

Sample Input

```
$10
$100
?
$10
$1
$50
$50
$1000
$1
```

Sample Output

```
$1 $10 $50
```

Group 5 Problem - Chain of Fools

Many of you have heard the story of Turing's bicycle: Seems the sprocket on the crank of the bicycle had a broken prong. Also his chain had one link that was bent. When the bent link on the chain came to hook up with the broken prong, the chain would fall off and Turing would stop and put the chain back on. But Turing, being who he was, could predict just when this was going to happen — meaning he would know how many pedal strokes it would be — and so would hop off his bike just before it happened and gently move the pedals by hand as the undesired coupling passed. Then he'd be merrily (we imagine) on his way. (A picture of the sprocket-chain set up is shown below.)
Your job here is to calculate the number of revolutions required in such a situation as Turing's: You'll be given the number of prongs on the front sprocket, the number of links on the chain, the location of the broken prong and the location of the bent link in the chain. The top prong is at location 0, then the next one forward on the sprocket is location 1 and so on until prong numbered s − 1. Prong s − 1 is the next prong that moves to the top of the sprocket as Turing pedals. Location of the links is similar: The link at the top of the sprocket is location 0 and so on forward until c − 1. The chain falls off when broken prong and bent link are both at location 0.

Input

Input for each test case will be one line of the form *s c p l*, where *s* is the number of prongs on the front sprocket ($1 < s < 100$) , *c* is the number of links in the chain ($200 > c > s$), *p* is the initial position of the broken prong, and l is the initial position of the bent link. The line 0 0 0 0 will follow the last line of input.

Broken prong and bent link will never both start at position 0.

Output

For each test case output a single one line as follows:

Case n: r m/s

if it requires r m/s revolutions to first fail, or

Case n: Never

if this can never happen.

Note that the denominator of the fraction will always be the number of prongs on the sprocket; the fraction will not necessarily be in lowest terms. Always print the values of r and m, even if 0.

Sample Input

```
40 71 32 23
20 40 4 24
40 71 8 33
20 40 3 17
0 0 0 0
```

Sample Output

```
Case 1: 1 8/40
Case 2: 0 16/20
Case 3: 25 32/40
Case 4: Never
```

Group 6 Problem - Condorcet Winners

A *Condorcet winner* of an election is a candidate who would beat any of the other candidates in a one-on-one contest. Determining a Condorcet winner can only be done when voters specify a ballot listing all of the candidates in their order of preference (we will call such a ballot a preference list). For example, assume we have 3 candidates — A, B and C — and three voters whose preference lists are ABC, BAC, CBA. Here, B is the Condorcet winner, beating A in 2 of the three ballots (ballots 2 and 3) and beating C in 2 of the three ballots (1 and 2).

The Condorcet voting system looks for the Condorcet winner and declares that person the winner of the election. Note that if we were only considering first place votes in the example above (as we do in most elections in the US and Canada), then there would be a tie for first place. There can be at most only one Condorcet winner, but there is one small drawback in the Condorcet system — it is possible that there may be no Condorcet winner.

Input

Input for each test case will start with a single line containing two positive integers b c, where b indicates the number of ballots and c indicates the number of candidates. Candidates are considered numbered 0,...,c−1. Following this first line will be b lines containing c values each. Each of these lines represents one ballot and will contain the values 0,...,c − 1 in some permuted order. Values of b and c will lie in the ranges 1...500 and 1...2500, respectively, and the line 0 0 will follow the last test case.

Output

For each test case output a single line containing either the candidate number of the Condorcet winner, or the phrase No Condorcet winner using the format given.

Sample Input

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

Sample Output

```
Case 1: 1
Case 2: No Condorcet winner
```