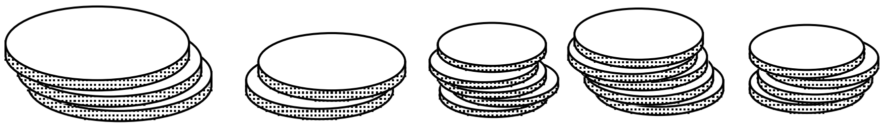
Ninth Annual
University of Oregon Programming Competition

Saturday, May 7, 2005

Change Maker



A local fifth grade class has been studying the mathematics of combinations. The students particularly like the problems that give an amount of money and ask whether that amount can be formed from a certain number of standard US coins. For example, they might be given the amount of 41¢ and be asked whether it can be formed with one coin, or two coins, or three coins, and so on, all the way up to 41 coins. The possible coins are the half dollar, quarter, dime, nickel, and penny. If the amount cannot be formed, they are to respond "IMPOSSIBLE"; otherwise, they should determine the numbers for each type of coin required. For example, 41¢ is impossible to make from one or two or three coins, but can be made from four coins consisting of a quarter, a dime, a nickel and a penny, and it can made in *two different ways* from five coins consisting of either a quarter, 3 nickels and a penny or four dimes and a penny.

In this problem, you are given an amount of money and a specified number of coins. Your job is to determine whether the amount can be formed from a combination of that many coins and, if so, report the particular combination of coins that has the most high-valued coins.

The first line of the input to your program will be an integer n indicating the number of coinage problems to follow. Each of the next n lines will contain a problem represented by a pair $A\ C$ of positive numbers with C an integer ≤ 100 and $A \leq 10.00$ representing dollars and cents expressed in standard form (with two places after the decimal point but, for input convenience, omitting an initial dollar sign "$").

For each pair $A\ C$, you should indicate whether $\$A$ can be formed with C coins. If that is impossible, the line should read

`IMPOSSIBLE to make $`A` with `C` coins.`

(Note the reinsertion of "$" for the output.) If it is possible, you should report the corresponding combination of C coins (as formatted in the samples). If there is more than one such combination, you should report the one that has the most high-valued coins; that is, use the one with the most number of half-dollars; if there is more than one combination with that maximum number of half-dollars, then report the one with the most number of quarters, etc.

Sample Input

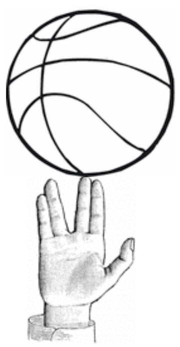
```
5
0.41 3
0.41 4
0.50 1
1.25 100
2.50 5
```

Sample Output

```
IMPOSSIBLE to make $0.41 with 3 coins
$0.41 can be made with 4 coins: 1 quarter 1 dime 1 nickel 1 penny
$0.50 can be made with 1 coin: 1 half
$1.25 can be made with 100 coins: 1 dime 4 nickels 95 pennies
$2.50 can be made with 5 coins: 5 halves
```

K'reBrax Craziness

Vulcan schools quickly caught the fever of Terran basketball. A packed season on Vulcan comes to a climax during the month of K'reBrax with the VCAA Tournament.



Now, the logical Vulcan sports fan would not tolerate a subjective selection of teams. Instead, eligible teams are precisely those that have won at least 78.85398% of their conference games.

Nor would the Vulcan mind accept the irrational practice of *seeding* to schedule the matches. Instead, matches are played sequentially (there is only one TV channel broadcasting VCAA games) between pairs of teams selected at random (via an unbiased random-number generator initiated by the incidence of cosmic rays) except that games are not played if their outcome can be logically deduced. For example, if it is known that Southern ShiKahr has defeated, or would defeat, the Vulcan Science Academy which has defeated, or would defeat, T'Khut A&M, then it is declared that Southern ShiKahr would defeat T'Khut A&M and so there is no need for such a match.

Furthermore, the VCAA Tournament does not terminate with the declaration of a winner but continues until it determines a complete ordering of the teams involved.

For this problem, you are to write a program that determines which numerical rankings can be decided from the games played so far. (The teams with these ranks can head home because they will not need to play another game.)

The first line of input to your program is an integer n, indicating the number of partial tournaments to follow. Then the first line of each tournament contains a pair of integers t g, separated by a single space, where $t \leq 100$ is the number of teams that were eligible in the current tournment and g is the number of games played so far. The next g lines give the outcomes of these games (not necessarily in chronological order), where

$$\alpha \;\; \beta$$

signifies that team α defeated team β. Teams are identified by the 2 or 3 letter initials of their school.

The output to your program should indicate, for each tournament instance, the set of ranks that can already be determined, with

$$r \; \alpha$$

signifying that team α has rank r. If the set is empty then your output should read

```
No ranks can be determined.
```

There should be one such pair per line and tournament instances should be separated by a blank line.

Sample Input

```
5
4 3
TAM UV
SS VSA
VSA TAM
3 1
SS VSA
3 2
SS VSA
SS TAM
4 3
SS VSA
TAM VSA
TAM UV
6 5
SS VSA
VSA TAM
TAM UV
SCC VSA
VSA VC
```

Sample Output

```
1 SS
2 VSA
3 TAM
4 UV

No ranks can be determined.

1 SS

No ranks can be determined.

3 VSA
```

A Mini-Interpreter

This problem involves writing an interpreter for L, a tiny, expression-oriented language with functions, conditionals, integer arithmetic (`+`, `-`, and `*` for addition, subtraction, and multiplication), and little else. Each line of an L program is either a function definition or an expression, except that the last line contains only the end-of-program marker "#".

Tokens (symbols, keywords, and positive integer constants) are separated by blanks. Function names are single upper-case letters, and variable names are single lower-case letters.

Expressions in L use prefix syntax. For example, `* + 5 2 9` evaluates to 63, but `* 5 + 2 9` evaluates to 55.

A function call consists of a function name (a single upper-case letter), followed by a sequence of expressions, followed by "." For example, what we might conventionally write as `3 + F(G(x), y+3) + 2` is expressed in L as

```
+ + 3 F G x . + y 3 . 2
```

A conditional expression `?` e_1 e_2 e_3 is evaluated as follows: First e_1 is evaluated. If the result is not zero, then e_2 is evaluated and its value becomes the value of the conditional expression. Otherwise, e_3 is evaluated and its value becomes the value of the conditional expression. Only one of e_2 or e_3 is evaluated.

A function definition is written `def` F v_1 v_2 $\ldots v_n$ `.` e , where F is any upper-case letter (the function name), the v_i are distinct lower case letters (formal parameters), and e is an expression to be evaluated when the function is called. Formal parameters (but no other variables) may appear in the expression. Recursive and mutually recursive functions are permitted.

The number of actual arguments in a function call F e_1 e_2 $\ldots e_n$ `.` must match the number of formal parameters of F. Function calls are interpreted using "pass by value" semantics, i.e., arguments are evaluated before binding to parameters. A function may not be called until it has been defined (but a call to the function may appear before its definition in another function definition, thus permitting mutual recursion).

There are only two scopes in an L program: A global scope of function definitions, and a local scope of variable bindings for the currently executing function (or an empty variable binding scope when evaluating an expression at the top level).

The input to your program will be a single, syntactically correct, L program with the last line signaled by '#'.

Your interpreter must read and execute each line of the L program. For each line that is an expression, your program should output the integer result on a line.

Sample Input

```
def F x .  ?  x * F - x 1 .  x 1
F 5 .
F + 3 2 .
F F 2 .  .
#
```

Sample Output

```
120
120
2
```

Sample Input

```
def D x .  + x x
def Q a .  D D a .  .
def F x .  - Q x .  D x .
D 12 .
Q 12 .
F 12 .
- F 24 .  F 48 .
#
```

Sample Output

```
24
48
24
-48
```

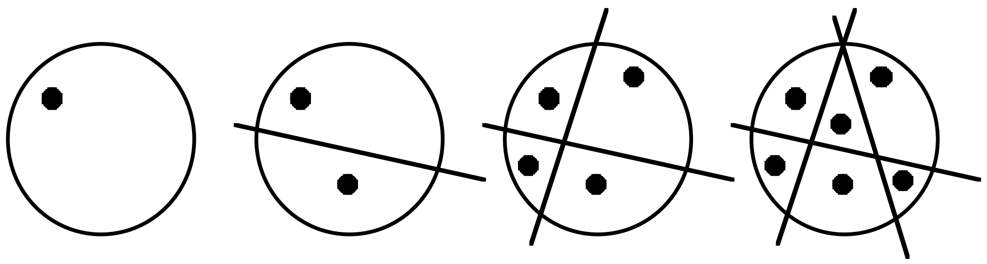
Sample Input

```
?  13 4 2
+ 3 ?  2 2 + 4 7
?  + 5 - 0 5 17 + + 33 33 33
#
```

Sample Output

```
4
5
99
```

Miss the Marbles



The two-person game of *Miss-the-Marbles* is played as follows. A fixed circle is drawn on the ground and a single marble is placed in the circle by the first player, $\mathcal{A}$, who then shoots another marble, which must go through the circle *missing* the marble inside. The path of that second marble thus divides the circle into two regions, one of which contains the original marble; player $\mathcal{B}$ then has to place a marble in the region that is empty. Player $\mathcal{B}$ then shoots a marble through the circle hoping to miss both the marbles inside. This path of this marble then creates one or two new empty regions into which $\mathcal{A}$ must put marbles. Continuing to shoot marbles in turn, the object is to avoid hitting any marbles inside the circle and thereby forcing your opponent to put a marble in each empty region.

The game continues until a shooter either hits a marble or misses the circle entirely, in which case the shooter's opponent wins all the marbles in the circle.

For this problem, you are to write a program that determines how many marbles are in the circle after the first m rounds, given the paths taken by the first m marbles shot successfully through the circle.

The first line of input to your program will be an integer n indicating how many game instances are to follow. Each game instance will begin with a single integer m on a line indicating how many marbles have been shot so far. The next m lines will each contain 4 integers $a\ b\ c\ d$, separated by single spaces, with $(a, b) \neq (c, d)$; this four-tuple indicates that a marble has been shot from point (a, b) to point (c, d). The paths so far will all have passed through the circle and will have missed hitting any marbles inside.

The circle will always have radius 1 and be centered at the origin $(0, 0)$.

For each of the n instances, the output (one per line) should be a single integer indicating the number of marbles in the circle at that point of the game.

Sample Input

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

Sample Output

```
2
3
4
6
```

Prime Power Towers

Donald Knuth introduced a notation to express the large numbers produced by iterated exponentials, namely,

$$p \uparrow\uparrow r \;=\; p^{p^{p^{\cdot^{\cdot^{p}}}}}$$

where there are r p's on the right hand side. Thus

$$p \uparrow\uparrow 1 = p \quad \text{and} \quad p \uparrow\uparrow (r+1) = p^{p\uparrow\uparrow r}.$$

This problem will involve computation of such exponential towers. However, these numbers could be of an unwieldy size, e.g., even $10 \uparrow\uparrow 3$ (commonly known as a *googol*) is already several orders of magnitude greater than the number of elementary particles in the universe. Fortunately, a certain cryptographic application only requires computation of

$$(p \uparrow\uparrow r) \pmod{m}$$

when p is prime and $m < p$.[*]

The first line of input to your program will be an integer n indicating how many problem instances will follow, one to a line. Each of the next n lines will have a triple,

$$p\ r\ m$$

of positive integers, separated by single spaces, where p is prime, $m < p < 1000$ and $r \leq 10$.

The output, for each of the n input triples $p\ r\ m$, should be $(p \uparrow\uparrow r) \pmod{m}$.

Sample Input

```
3
11 1 8
5 2 3
3 3 2
```

Sample Output

```
3
2
1
```

[*]The notation $N \pmod{m}$ signifies the remainder when N is divided by m.