2013 Consortium for Computing Sciences in Colleges Programming Contest
Saturday, November 16th
Furman University
Greenville, SC

There are eight (8) problems in this packet. Each team member should have a copy of the problems. These problems are NOT necessarily sorted by difficulty. You may solve them in any order.

Remember input/output for the contest will be from `stdin` to `stdout`. `stderr` will be ignored. Do not refer to or use external files in your source code. Extra white space at the end of lines is ignored, but extra white space at the beginning or within text on a line is not ignored. An extra blank line of output is ignored, but blank lines at the beginning or between lines of text are not ignored.

Have A Lot Of Fun & Good Luck! ☺

Problem 1. Ordered Lists

Problem 2. Good or Evil?

Problem 3. River Dice

Problem 4. Single Instruction Assembly

Problem 5. Old Calculator

Problem 6. Crossing the Mississippi

Problem 7. Who Got What?

Problem 8. Run-Length Encoding

Problem 1

Ordered Lists

For this problem, you will write a program that scans a list of integers to verify whether or not it is ordered. To make things simple, each of the lists will always have 10 integer values. A list is said to be in *ascending orde*r if all 10 values are listed in order from least to greatest. A list is said to be in *descending order* if all 10 values are listed in order from greatest to least. For this problem, there must be at least two different values in the list for it to be ordered. That is, a list with all duplicate values is not ordered.

Input

The first line of input contains a single integer n, ($1 \leq n \leq 1000$), which is the number of test cases that follow. Each test case consists of a single line containing exactly ten integers. Each of the ten integers will always be separated by a single space.

Output

For each test case, generate one line of output in the exact format below indicating the list number, and whether the list is in ascending order, descending order, or not ordered. Do not forget the ending period.

Sample Input

```
5
1 2 3 4 5 6 7 8 9 10
50 39 28 15 9 5 3 2 1 0
5 5 5 5 5 5 5 5 5 5
5 6 7 5 4 2 5 1 5 0
5 6 7 8 9 -1 -2 -3 -4 -5
```

Output Corresponding to Sample Input

```
List 1 is in ascending order.
List 2 is in descending order.
List 3 is not ordered.
List 4 is not ordered.
List 5 is not ordered.
```

Problem 2

Good or Evil?

You are trying to figure out whether a character is good or evil. Well, it's easy to tell! You just have to count up the number of g's and e's in their name. If they have more g's, they are good, if they have more e's, they are bad. Think about it, the greatest hero of them all, Algorithm Crunching Man is good since he has two g's and no e's. What about Tux (the *Linux* Penguin)? Well, the name Tux has no g's and no e's so he must be neutral. Any name with the same number of g's and e's is deemed neutral. Be sure to ignore case in your solution (i.e., count uppercase and lowercase g's and e's the same).

Input
The first line of input contains a single integer n, ($1 \leq n \leq 1000$), which is the number of test cases that follow. Each test case consists of a single line of input containing a string. Each string is of length n, ($1 \leq n \leq 80$).

Output
For each test case, you should generate one line of output with the name followed by a single space, followed by `is`, a single space, and then followed by either `GOOD`, `EVIL`, or `NEUTRAL` based on the relation of g's to e's. Each result should appear on a separate line.

Sample Input
```
5
Hades
Glinda the Good Witch
Simba
Maleficent
Cruella De Vil
```

Output Corresponding to Sample Input
```
Hades is EVIL
Glinda the Good Witch is GOOD
Simba is NEITHER
Maleficent is EVIL
Cruella De Vil is EVIL
```

Problem 3

River Dice

A river roll is a sequence of single dice rolls that has an interesting property: dice rolls after the first are always greater than or equal to the previous roll. For instance, if you roll two six-sided dice and get a 1 and then a 4 you've got a river roll. However, if you roll a 5 and then a 4 the sequence is not considered a river roll.

Your task is to calculate the *exact* probability of getting a river roll for a set of *n* fair dice with *k* sides. For instance, the previous case with two six-sided dice has $6^2 = 36$ possible outcomes, but only 6 + 5 + 4 + 3 + 2 + 1 = 21 possible non-decreasing sequences. This gives a probability of 21/36 = 7/12. It is worth noting that any sequence with just a single die is always non-decreasing, but a sequence with no dice is not.

For each of these lines, you should print out the probability as the simplest fraction possible.

Input

The input will consist of a positive integer *c*, which is the number of cases to follow. The next *c* lines will contain two integers *n* and *k*, the number of dice and the number of sides respectively. You are guaranteed that ($0 \leq n \leq 16$) and ($4 \leq k \leq 12$).

Output

For each of these lines, you should print out the probability as the simplest fraction possible in the format shown below.

Sample Input

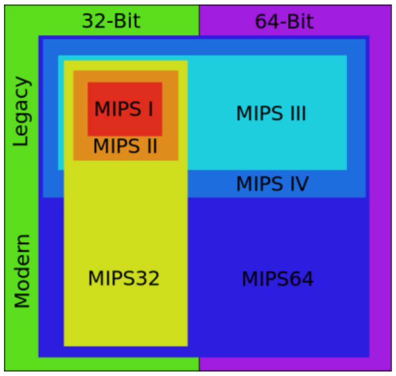
```
3
0 6
1 6
2 6
```

Output Corresponding to Sample Input

```
0/1
1/1
7/12
```

Problem 4

Single Instruction Assembly



You've finally done it! You created the world’s most efficient computer. In fact, it is so efficient that it doesn't even need op codes in its assembly language. This is because it only has a single instruction. The computer is a 12-bit machine, with 16 memory locations. So, a single memory slot holds three hex digits, and can be addressed by a single hex digit. If we take the contents of a memory location, we can call the high part a, the middle part b, and the low part c.

Using this notation, the single instruction works like this: The computer takes the contents of memory slot `a`, subtracts them from the contents of memory slot b and then puts them back in memory slot `b`. It then checks the value of memory slot b. If this value is negative (the highest of the 12 bits is set to 1), it moves the instruction pointer to the memory location pointed to by `c`. Otherwise, it moves the instruction pointer to the next slot, wrapping around to the beginning if necessary. It is important to note how subtraction behaves. The instruction pointer always starts at the first memory location. For instance, if I subtract `0 - 2`, the math should wrap around (modulus style) to `fe`. If I subtract `0 - 1`, the math should wrap around (modulus style) to `ff`.

Because this computer is so simple, it doesn't really have a way to know when to stop. So, we'll have to specify a number of steps before we stop.

Input

The first line of input will have a positive integer *n*, telling how many cases follow. The next *n* lines will have a sequence of 48 hex digits representing the memory (starting state of the computer), followed by an integer *k* ($0 \leq k \leq 256$) that tells how many steps to take before terminating.

Output

When the program completes, you should print out the final state of the computer, in the same format as seen below.

Sample Input

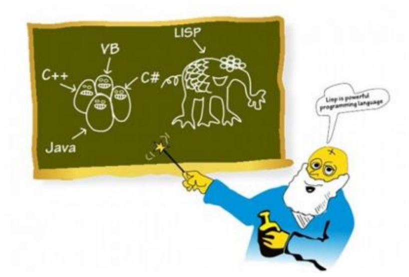
```
2
123001001000000000000000000000000000000000000000 2
123001000330000000000000000000000000000000000000 2
```

Output Corresponding to Sample Input

```
000001000000000000000000000000000000000000000000
123001fff000000000000000000000000000000000000000
```

Problem 5

Old Calculator



You just stumbled upon an old calculator in your basement with an oddly familiar syntax. The calculator only supports addition, subtraction, multiplication, and exponentiation and processes them in an odd order: it puts the operator first, and the operands second. Furthermore, in order to nest operations, you surround them with parentheses. For instance `7 + (2 * 3)` would be entered as `(+ 7 (* 2 3))`. The calculator, being old, is somewhat finicky and demands the operator always have no space before, but always a space after it. Furthermore, the calculator does not allow a space after the last operand just before a right parenthesis. So, `( + 2 1)`, `(+2 1)`, and `(+ 2 1 )` are all invalid according to this calculator.

After tinkering around, you've noticed that the calculator does not support explicit use of negatives, i.e. you couldn't have `(+ -7 2)`, so you have to replace it with `(+ (- 0 7) 2)`.

The calculator supports addition (+), subtraction (–), multiplication (`*`), and exponentiation (`^`). All of these operators are binary. Division is excluded. Multiple spaces are not allowed between operators and/or operands.

Despite the calculators' age, it appears to be in good working order. However, you've taken it upon yourself to *simulate* this calculator in your language of choice by writing an interpreter for the calculator's language.

Input

Input will start with an integer *n* on a single line, telling how many cases to follow. The next *n* lines will contain a single calculator statement on each line. All statements will fit on a single line and will not exceed 200 characters. A single integer by itself is not a valid calculator statement.

Output

For each line, you should print the single integer that the statement evaluates to. All computations will fit into a 32-bit integer.

Sample Input

```
2
(^ 2 10)
(+ (* 2 1) (- 8 2))
```

Output Corresponding to Sample Input

```
1024
8
```

Problem 6

Crossing the Mississippi

The Mississippi River runs through the heartland of America. You and your friends are planning a road trip where you will visit several cities in sequence. During the trip, you will want to know how many times you will have to cross this mighty river. For example, if your trip contains four cities, it is possible that you could cross the river up to three times.

But, you have not yet finalized your itinerary. You are considering several possible trips. A trip consists of a list of cities. Technically, the trip begins with the first city on the list, and ends with the last city listed. In particular, we do not count any travel from your home to the first city, or from the last city on a list back to your home.

For the purpose of this program, assume the following:

- All cities in Minnesota (MN), Iowa (IA), Missouri (MO), Arkansas (AR), and Louisiana (LA) are located west of the Mississippi River.
- All cities in Wisconsin (WI), Illinois (IL), Kentucky (KY), Tennessee (TN), and Mississippi (MS) are located east of the Mississippi River.
- Your road trip will only contain cities in these states on either side of the river.
- Within a trip, the cities must be visited in the order indicated.

Input

The first line of the input will say `<t> trips`, where `t` is an integer and $2 \le t \le 200$.
The rest of the input will itemize the individual trips. Each trip will begin with a line of this format:

```
Trip <i> has <c> cities.
```

where `c` is an integer and $2 \le c \le 200$. The trip numbers are numbered consecutively from 1.
The next `c` lines will each contain the name of a city. The format of a city name is:

```
<city name>, <state abbreviation>
```

Note that the city name could contain several words, but it will always be followed by a comma, a space and then the 2-letter state abbreviation. Both letters in this abbreviation will be capitalized. Blank space characters could follow the state abbreviation before the end of the line.

Output
Your program needs to indicate the number of river crossings on each trip, one trip per line of output. Each line of your output must be formatted like this:

```
Trip <i> crosses the Mississippi <n> times.
```
Where `i` is the trip number, and `n` is the number of river crossings your program has calculated. Note that if the number of river crossings is exactly 1, then the word "times" at the end of the sentence must be in the singular, "time."

Sample Input
```
2 trips
Trip 1 has 4 cities.
La Crosse, WI
Effingham, IL
St. Louis, MO
Jackson, TN
Trip 2 has 3 cities.
Mountain Home, AR
Rolla, MO
Davenport, IA
```

Output Corresponding to Sample Input
```
Trip 1 crosses the Mississippi 2 times.
Trip 2 crosses the Mississippi 0 times.
```

Problem 7

Who Got What?

During the holiday season, many groups and families get together and exchange gifts. Some groups play a right/left word game as part of the exchange. The instructions for one version of the game are as follows:

Get in a circle, pass gift packages right or left as cue words are read in the story.

> Many years ago high on a mountain, there lived a family named WRIGHT. They had many cousins whose names were also WRIGHT. Every Christmas the clan gathered to have a party and no one was LEFT out. Sally WRIGHT was in charge of the guest list and she LEFT nothing to chance. This party was planned and things had to be just RIGHT. Mary WRIGHT had LEFT home early to do her shopping, but LEFT her gift list at home. Now this created a problem RIGHT from the start for her. John WRIGHT had gone to cut a tree for the big affair. It had to be the RIGHT size to stand on the LEFT side of the fireplace. ...

You are to write a program that simulates an exchange for a given story and set of cue words. The program's output will tell whose gift is received by person 1. One simulates the exchange by processing the story and whenever a cue word is encountered *that is a complete word*, all gifts are passed offset number of places (for the offset associated with the cue word). For an offset of 1, person 2 passes to person 3, while an offset of -1 causes person 2 to pass to person 1 (when there are at least three people participating in the exchange).

Input

The first line will be the number of exchanges. Each exchange will have the following format. Its first line will contain the number *N* of people participating in the exchange, a space, and then number *K* of cue words. The next *K* lines will have an offset, an integer, followed by a space and then a cue word. Each cue word will consist of one or more alphabetic characters. You should ignore case altogether. (i.e., "`one`", "`One`", "`ONe`", & "`ONE`" are all the same.)

The next line will contain the phrase "`Story #`" followed (immediately) by a positive integer and then a colon. The story will then follow. **Note that consecutive exchanges will be separated by at least two blank lines.**

Output
If person 1 does not receive his/her own gift, then output "Person 1 receives the gift from person *m*." where *m* is the number of the person whose gift person 1 receives. However, if person 1 receives his/her own gift, then output "Each person receives his/her own gift."

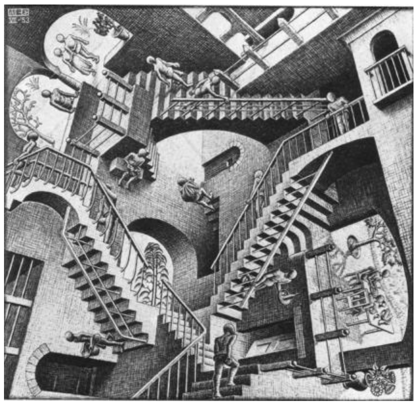
Sample Input
```
2
3 1
1 one
Story #1: This is one short story.


20 5
-1 left
1 right
1 write
1 wright
-2 one
Story #2:
Many years ago high on a mountain, there lived a family named WRIGHT.
They had many cousins whose names were also WRIGHT. Every Christmas
the clan gathered to have a party and no one was LEFT out. Sally
WRIGHT was in charge of the guest list and she LEFT nothing to chance.
This party was planned and things had to be just RIGHT. Mary WRIGHT
had LEFT home early to do her shopping, but LEFT her gift list at
home. Now this created a problem RIGHT from the start for her. John
WRIGHT had gone to cut a tree for the big affair. It had to be the
correct size to stand on the LEFT side of the fire place
```

Output Corresponding to Sample Input
```
Person 1 receives the gift from person 3.
Each person receives his/her own gift.
```

Problem 8

Run-Length Encoding



Run-length encoding (RLE) is a very simple form of data compression in which *runs* of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. This is most useful on data that contains many such runs: for example, simple graphic images such as icons, line drawings, and animations. It is not useful with files that don't have many runs as it could greatly increase the file size.

Your job is to write a compress program that can be used to encode an input sequence using run-length encoding. Encoding is used only if it shortens the original. **This means that three or fewer data values are not encoded.** First, we choose one special, unused byte value (`ff`) to indicate that a run-length code follows. Note that `ff` is guaranteed to never appear in the input. Then, the run-length encoding algorithm goes like this:

- Where the same value occurs more than three times in succession, substitute the following 3 bytes, in order:
  - The special run-length code indicator (`ff`) followed by a space
  - The byte value that is repeated followed by a space; and
  - The number of times that the value is repeated in decimal format followed by a space. Values smaller than 10 should be preceded with a leading 0.

For example, suppose we wish to compress a sequence of byte values using run-length encoding. The sequence "`93 93 93 93 92 91 91 94 94 94 94 94 95 95 95 73 73 73 73 73 73 73`" would output “`ff 93 04 92 91 91 ff 94 05 95 95 95 ff 73 07`”.

Input
The file contains a positive integer *n* and a sequence of *n* byte values, one sequence per line. Each sequence is of length *n*, ($2 \leq n \leq 2048$). Each byte value is a hexadecimal number.

Output
For each test case, generate one line of output for the encoded sequence.

Sample Input

```
3
22 23 24 24 24 24 24 24 24 25 26 26 26 26 26 26 25 24
01 01 01 01 01 01 01 01 01 01 01 04 04 02 02 03 03 03 03 04 05 06 06 07
07 07 02 03 03 05 05 06 06 05 05 04 04
```

Output Corresponding to Sample Input

```
22 23 ff 24 07 25 ff 26 06 25 24
ff 01 11 04 04 02 02 ff 03 04 04 05 06 06 07
07 07 02 03 03 05 05 06 06 05 05 04 04
```