2001 Stanford Local Programming Contest

October 13, 2001

Read these guidelines carefully!

Rules

1. You may use resource materials such as books, manuals and program listings. You may *not* use any machine-readable versions of software, data or existing electronic code. All software submitted during the contest *must* be typed in during the contest. No cutting and pasting of code is allowed.
2. Students may not collaborate in any way with each other or anybody else, including people contacted via Internet.
3. You are expected to adhere to the honor code. You are still expected to conduct yourself according to the rules, even if you are not in Gates B02.

Guidelines for submitted programs

1. All programs must be written in C, C++, or Java. To judge the programs, I will compile programs in the following way:
   - `.c`: using `gcc -lm`
   - `.cc`: using `g++ -lm`
   - `.java`: using `javac`

   I will use the Leland system to do this, so if your programs compile on a Leland host, they should compile for me. Please use only the listed filename extensions.
2. **Java users: Please place your `public static void main()` function in `public class Main`.** I want to be able to run your program with the command `java Main`.
3. Your solution may have as many files as necessary. However, please use the same language (C, C++, or Java) for a given run. Make sure each problem's code files are in a separate directory.
4. All programs should accept their input on **stdin** and produce their output on **stdout**. They should be batch programs in the sense that they do not require human input other than what is piped into stdin.
5. Be careful to follow the output format described in the problem. I will be judging programs based on a `diff` of your output with the correct solution.

How will the contest work?

1. From 12:00 to 1:00 students will pick a computer, set up their workspace and complete a test problem. The purpose of this test problem is to send me information about who you are; you won't be registered until you do this. This will also make sure the submission script is working for you.
2. At 1:00 the contestants will gather in the **outside Gates B02** and I will hand out the problems. You should then return to your workspace and begin solving the problems.
3. The test problem description explains how to submit your solutions. **Make sure that you run the submission script from the correct directory, and that only the files for your solution are in that directory!**

4. For every run, your solution will either be accepted or rejected for one of the following reasons: *compiler error, run time error, time limit exceeded, wrong answer, presentation error*. The time limit for a run is **one minute**.

5. Watch your **email on Leland**. I will be sending out any necessary messages as well as notification if your programs are accepted or rejected via email, and the emails will go to

   `your_leland_id@leland.stanford.edu`.

6. If you want to follow the progress of the contest, go to the **standings web page**. A link to this page exists on the main contest page, http://www-db.stanford.edu/~cooperb/acm.

7. At 4:00 the contest will end. No more submissions will be accepted. The contestants will be ranked by the number of solved problems. Ties will be broken based on the total time used for correct solutions (the time elapsed since 1:00 plus 20 minutes per rejected solution).

8. The top six contestants will advance to the regionals, subject to the constraint of a maximum of one graduate student per three person team. Contestants 7-9 will form the alternate team, which will advance to the regionals if there is enough room there.

If you have any questions, come talk to me in **Gates B02**, or send email to **cooperb@leland**.

0 Test Problem

0.1 Description

This is a test problem to make sure you can compile and submit programs. Your program for this section will print your leland user id, name and whether you are a graduate student or an undergraduate.

You must submit your solutions via the Leland system and the submission script I have created. You can develop your programs on whatever platform you want, but the submission must be via Leland.

Your solution to a given problem should be in its own directory, separate from the other problems. For example, you may have a `p0` directory, a `p1` directory, and so on. When you submit your solution, everything in this directory will be tarred and sent to the judge, so be careful that the right things are in the directory.

To submit a solution, run the `~cooperb/submit` script with a single argument: the problem you are submitting. For example, to submit problem 0, type:

```
$ ~cooperb/submit 0
```

Make sure you are in the directory with the solution for that program.
C/C++ users: Make sure only one of your C/C++ code files has a `main()` method.
Java users: Make sure there is a `public static void main()` method in `public class Main`.

When you submit your solution, the judge will receive an email and will judge your solution as soon as he can (be patient!) Once this is finished, you will receive an email stating that you have completed the problem or that you have not, and a reason why not. You can resubmit rejected solutions as many times as you like (though incurring a 20 minute penalty for each rejected run of a problem you eventually get right). Once you have submitted a correct solution, future submissions of that problem will be disregarded.

0.2 Input

This program will take no input.

0.3 Output

The program should output your leland user id, name and student status. This information should be separated by commas, and all on the same line. The line should be terminated with a newline. For example:

```
cooperb,Brian Cooper,Graduate
```

1 Scenario sequences

1.1 Description

Computer simulations test systems under various scenarios. In this way, we can determine the expected behavior of a complex system over a long period of time without having to build the system and wait for it to run. Instead, we can model the system, find scenarios where it performs poorly, and fix the system to deal with these scenarios.

In a simulator, therefore, it is important to be able generate a set of scenarios for testing. Often, a scenario has multiple properties, and each property can take on one of a set of values. For example, we may want to test a simulated CPU under the following conditions:

TEMPERATURE=96degrees
CLOCKSPEED=1.2Ghz
BUSSPEED=100Mhz

In order to make it easy for a researcher to input a scenario into the simulator, property lists (such as the one above) are often stored in ASCII files and read in at simulation start time. It is even easier for the researcher if it is possible to specify all scenarios in a single test file. Then, however, there must be a module to construct all of the scenarios from the specification in the file. Your job is to write that module.

1.2 Input

Your program will read in a property list in the following form:

Property1=Value1
Property2=Value2
...

Your program is to ignore any whitespace. All property-value pairs will be terminated by a newline. The list of property-value pairs will be terminated by

END

on its own line. Each property may appear multiple times, for example "TEMPERATURE" may appear 3 times, and not necessarily contiguously. For example:

```
TEMPERATURE=96degrees
TEMPERATURE=98degrees
CLOCKSPEED=1.3Ghz
CLOCKSPEED=1.2Ghz
CLOCKSPEED=1.5Ghz
END
```

1.3 Output

Your program should output a comma-delimited sequence of all possible scenarios, where a scenario is a choice of one listed value for each listed property. On the first line of your output should be a comma delimited list, terminated by a newline, of the property names, in the order they first appeared in the input (the *input appearance order*):

P1,P2,P3...Pn

After that, each scenario should be printed. The values of the scenario should be listed in the same order as the property names in the first line of the output:

V1,V2,V3...Vn

The scenarios themselves should be sorted first on the input appearance order of V1, then on the input appearance order of V2, and so on, up to Vn. There should be no whitespace in output file at all, except for newlines. There should be no duplicate scenarios. For example:

```
TEMPERATURE,CLOCKSPEED
96degrees,1.3Ghz
96degrees,1.2Ghz
96degrees,1.5Ghz
98degrees,1.3Ghz
98degrees,1.2Ghz
98degrees,1.5Ghz
```

2 Kin

2.1 Description

Expression trees, B and B* trees, red-black trees, quad trees, PQ trees; trees play a significant role in many domains of computer science. Sometimes the name of a problem may indicate that trees are used when they are not, as in the Artificial Intelligence planning problem traditionally called the *Monkey and Bananas problem*. Sometimes trees may be used in a problem whose name gives no indication that trees are involved, as in the *Huffman code*.

This problem involves determining how pairs of people who may be part of a "family tree" are related.

Given a sequence of *child-parent* pairs, where a pair consists of the child's name followed by the (single) parent's name, and a list of query pairs also expressed as two names, you are to write a program to determine whether the query pairs are related. If the names comprising a query pair are related the program should determine what the relationship is. Consider academic advisees and advisors as exemplars of such a single parent genealogy (we assume a single advisor, i.e., no co-advisors).

In this problem the child-parent pair $p \quad q$ denotes that p is the child of q. In determining relationships between names we use the following definitions:

- p is a *0-descendent* of q (respectively *0-ancestor*) if and only if the child-parent pair $p \quad q$ (respectively $q \quad p$) appears in the input sequence of child-parent pairs.
- p is a *k-descendent* of q (respectively *k-ancestor*) if and only if the child-parent pair $p \quad r$ (respectively $q \quad r$) appears in the input sequence and r is a $(k-1)$-descendent of q (respectively p is a $(k-1)$-ancestor of r).

For the purposes of this problem the relationship between a person p and a person q is expressed as exactly one of the following four relations:

1. child — grand child, great grand child, great great grand child,

   By definition p is the "child" of q if and only if the pair $p \quad q$ appears in the input sequence of child-parent pairs (i.e., p is a 0-descendent of q); p is the "grand child" of q if and only if p is a 1-descendent of q; and

   $$p \text{ is the ``}\underbrace{\text{great great} \ldots \text{great}}_{n \text{ times}}\text{grand child'' of } q$$

   if and only if p is an $(n+1)$-descendent of q.

2. parent — grand parent, great grand parent, great great grand parent, *etc.*

   By definition p is the "parent" of q if and only if the pair $q \quad p$ appears in the input sequence of child-parent pairs (i.e., p is a 0-ancestor of q); p is the "grand parent" of q if and only if p is a 1-ancestor of q; and

   $$p \text{ is the ``}\underbrace{\text{great great} \ldots \text{great}}_{n \text{ times}}\text{grand parent'' of } q$$

   if and only if p is an $(n+1)$-ancestor of q.

3. cousin — 0^{th} cousin, 1^{st} cousin, 2^{nd} cousin, *etc.*; cousins may be once removed, twice removed, three times removed, *etc.*

   By definition p and q are "cousins" if and only if they are related (i.e., there is a path from p to q in the implicit undirected parent-child tree). Let r represent the least common ancestor of p and q

(i.e., no descendent of r is an ancestor of both p and q), where p is an m-descendent of r and q is an n-descendent of r.

Then, by definition, cousins p and q are "k^{th} cousins" if and only if $k = \min(n, m)$, and, also by definition, p and q are "cousins removed j times" if and only if $j = \mid n - m \mid$.

4. sibling — 0^{th} cousins removed 0 times are "siblings" (they have the same parent).

2.2 Input

The input consists of parent-child pairs of names, one pair per line. Each name in a pair consists of lower-case alphabetic characters or periods (used to separate first and last names, for example). Child names are separated from parent names by one or more spaces. Parent-child pairs are terminated by a pair whose first component is the string "*no.child*". Such a pair is NOT to be considered as a parent-child pair, but only as a delimiter to separate the parent-child pairs from the query pairs. There will be no circular relationships, i.e., no name p can be *both* an ancestor and a descendent of the same name q.

The parent-child pairs are followed by a sequence of query pairs in the same format as the parent-child pairs, i.e., each name in a query pair is a sequence of lower-case alphabetic characters and periods, and names are separated by one or more spaces. Query pairs are terminated by end-of-file.

There will be a maximum of 300 different names overall (parent-child and query pairs). All names will be fewer than 31 characters in length. There will be no more than 100 query pairs.

For example:

```
alonzo.church oswald.veblen
stephen.kleene alonzo.church
dana.scott alonzo.church
martin.davis alonzo.church
pat.fischer hartley.rogers
mike.paterson david.park
dennis.ritchie pat.fischer
hartley.rogers alonzo.church
les.valiant mike.paterson
bob.constable stephen.kleene
david.park hartley.rogers
no.child no.parent
stephen.kleene bob.constable
hartley.rogers stephen.kleene
les.valiant alonzo.church
les.valiant dennis.ritchie
dennis.ritchie les.valiant
pat.fischer michael.rabin
```

2.3 Output

For each query-pair p q of names the output should indicate the relationship p *is-the-relative-of* q by the appropriate string of the form

- child, grand child, great grand child, great great ... great grand child
- parent, grand parent, great grand parent, great great ... great grand parent

- sibling
- n cousin removed m
- no relation

If an m-cousin is removed 0 times then only *m cousin* should be printed, i.e., *removed 0* should NOT be printed. Do not print *st, nd, rd, th* after the numbers.

For example:

```
parent
sibling
great great grand child
1 cousin removed 1
1 cousin removed 1
no relation
```

3 Up in the sky

3.1 Description

On a clear moon-less night, you can see millions of stars glimmering in the sky. Faced with this overwhelming number, the Greeks started nearly 2,000 years ago to bring some order to the chaos. They identified groups of stars, called constellations, and gave them names, mostly from the Greek mythology, that are still in use today. Examples are "Ursa Minor", "Pisces", "Cancer", and many others.

Given a sketch of the constellation, it is not easy for the amateur to actually find the constellation in the sky. Moreover, simple constellations, such as "Triangulum" (triangle,) which consists of only three stars, may appear several times in the sky. Again, singling out the "correct" occurrence is not easy. Traditionally, maps were printed for just this purpose. But in this problem, we will see how the computer can help us find constellations in the sky.

You will be given a star map; for simplicity this will be a collection of points in the plane, each having a certain brightness associated with it. Then, given a constellation, also as a set of points in the plane, you are to determine:

- the number of occurrences of the constellation in the star map, and
- the position of the brightest occurrence, if one exists. (The rationale behind this is as follows: if a constellation seems to appear several times in the sky, the brightest one is most likely to be the real one, since it is the most eye-catching one.)

An occurrence is a subset of stars from the map that forms a (possibly) arbitrarily rotated and/or scaled copy of the stars in the constellation. The brightness of an occurrence is the average brightness of the stars it consists of, i.e. the sum of individual brightnesses divided by the number of stars in the constellation.

3.2 Input

The input file contains the descriptions of several star maps. Each map starts with a line containing a single integer n, specifying the number of stars in the map ($1 \leq n < 1000$). The following n lines contain three integers each, namely the x- and y-coordinates and the brightness of every star. The larger the value, the brighter the star shines.

The next line contains a single integer m, the number of constellations to follow ($1 \leq m < 50$). Each constellation description starts with a line containing an integer s_i, the number of stars in constellation i, and a string N_i, the name of the constellation. (N_i will consist of no more than 40 characters and contain no blanks.) The following s_i lines then contain the coordinates of the constellation, again as x/y-pairs.

A blank line separates the star map from the next map. The input file ends with an empty map (having $n = 0$), which should not be processed. Since all star coordinates are integer numbers, you can easily rule out any rotated or scaled constellation whose points do not fall on integer coordinates.

For example:

```
6
1 2 1
2 1 4
2 4 3
3 2 1
4 1 5
```

```
4 3 2
2
3 Triangulum
1 1
3 1
2 4
4 Cancer
1 3
4 3
6 1
7 5

0
```

3.3 Output

For each star map first output the number of the map ('Map #1', 'Map #2', etc.) on a line of its own. For each constellation, in the same order as in the input, output first its name and how many times it occurs in the map on one line, as shown in the output sample. If there is at least one occurrence, output the position of the brightest occurrence by listing the positions of the stars that form the brightest occurrence. The star positions have to be printed in ascending x-order. Positions having the same x-coordinates must be sorted in ascending y-order. If there are several equally bright solutions, output only one of them. Adhere to the format shown in the sample output. Output a blank line before each constellation and a line of 5 dashes ('-----') after every star map.

```
Map #1

Triangulum occurs 2 time(s) in the map.
Brightest occurrence: (1,2) (4,1) (4,3)

Cancer occurs 0 time(s) in the map.
-----
```

4 0!

4.1 Description

Factorials are fun!

$10! = 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 3628800$

However, I've noticed that as I compute larger and larger factorials, there are more and more zeroes on the end (when written down in decimal notation). I want to know, just how many?

4.2 Input

The input to your program is a positive integer d, followed by d data sets. Each data set consists of a positive integer n, with $n < 10000$. For example:

```
3
2
6
10
```

4.3 Output

The output for each n should be $z(n)$, the number of trailing zeroes in the decimal expansion of $n!$. That is, $10^{z(n)}$ divides evenly into $n!$, but $10^{z(n)+1}$ does not. For example:

```
0
1
2
```

5 The oldest photocopier

5.1 Description

Before the invention of book-printing, it was very hard to make a copy of a book. All the contents had to be re-written by hand by so called scribers. The scriber had been given a book and after several months he finished its copy. One of the most famous scribers lived in the 15th century and his name was Xaverius Endricus Remius Ontius Xendrianus (Xerox). Anyway, the work was very annoying and boring. And the only way to speed it up was to hire more scribers.

Once upon a time, there was a theater ensemble that wanted to play famous Antique Tragedies. The scripts of these plays were divided into many books and actors needed more copies of them, of course. So they hired many scribers to make copies of these books. Imagine you have m books (numbered $1, 2, ..., m$) that may have different number of pages ($p_1, p_2, ..., p_m$) and you want to make one copy of each of them. Your task is to divide these books among k scribes, $k \leq m$. Each book can be assigned to a single scriber only, and every scriber must get a continuous sequence of books. That means, there exists an increasing succession of numbers $0 = b_0 < b_1 < b_2 ... < b_{k-1} \leq b_k = m$ such that i_{th} scriber gets a sequence of books with numbers between $b_{i-1} + 1$ and b_i. The time needed to make a copy of all the books is determined by the scriber who was assigned the most work. Therefore, our goal is to minimize the maximum number of pages assigned to a single scriber. Your task is to find the optimal assignment.

5.2 Input

The input consists of N cases. The first line of the input contains only the positive integer N. Then follow the cases. Each case consists of exactly two lines. On the first line, there are two integers m and k, $1 \leq k \leq m \leq 500$. On the second line, there are integers $p_1, p_2, ...p_m$ separated by spaces. All these values are positive and less than 10000000. For example:

```
2
9 3
100 200 300 400 500 600 700 800 900
5 4
100 100 100 100 100
```

5.3 Output

For each case, print exactly one line. The line must contain the input succession $p_1, p_2, ...p_m$ divided into exactly k parts such that the maximum sum of a single part should be as small as possible. Use the slash character ('/') to separate the parts. There must be exactly one space character between any two successive numbers and between the number and the slash. If there is more than one solution, print the one that minimizes the work assigned to the first scriber, then to the second scriber etc. But each scriber must be assigned at least one book. For example:

```
100 200 300 400 500 / 600 700 / 800 900
100 / 100 / 100 / 100 100
```