New Zealand Programming Contest 2006

5 August 2006

Preamble

Please note the following important details relating to input and output:

- Read all input from the keyboard, i.e., use `stdin`, `System.in`, `std::cin`, or equivalent. Input will be redirected from a file to form the input to your submission.
- Write all output to the screen, i.e., use `stdout`, `System.out`, `std::cout`, or equivalent. Do not write to `stderr`. Do **not** use, or even include, any module that allows direct manipulation of the screen, such as `conio`, `Crt`, or anything similar.

  Output from your program is redirected to a file for later checking. Use of direct I/O means that such output is not redirected and hence cannot be checked. This could mean that an otherwise correct program is rejected!
- Unless otherwise stated, all *integers* will fit into a standard 32-bit computer word. If more than one integer appears on a line, they will be separated by white space, i.e., spaces or tabs.
- An *uppercase letter* is a character in the sequence '`A`' to '`Z`'. A *lowercase letter* is a character in the sequence '`a`' to '`z`'. A *letter* is either a lowercase letter or an uppercase letter.
- Unless otherwise stated, a *word* or a *name* is a continuous sequence of letters, and a *string* is a continuous sequence of visible characters.
- Unless otherwise stated, words and names will contain no more than 60 characters, and strings will contain no more than 250 characters.
- If it is stated that 'a line contains no more than n characters', this does not include the character (or characters) specifying the end of line.
- All input files are terminated by a 'sentinel' line, followed by an end of file marker. This line should not be processed.

Please also note that the filenames of your submitted programs may need to follow a particular naming convention, as specified by the contest organisers at your site.

Problem A

~~Problem H~~

Octopus Numbers

10 points

Certain rows of ripple patterns in ocean-floor sand, limited to areas of the sea with very weak currents and low numbers of bottom-crawling creatures, have long puzzled oceanographers. In a recent discovery certain to revolutionise the field of animal linguistics, marine biologists have realised that these patterns are made by octopuses and represent a numbering system. Though as yet they can only speculate as to what the octopuses might be counting, they have succeeded in decoding the numbering system.

The digits used by the octopuses and the ripple patterns that represent them are quite unusual by land-based standards. Researchers have agreed to use the following typewriter symbols to represent corresponding similarly-shaped ripple patterns, and have determined their numeric values, as follows:

`-` represents 0	`>` represents 5
`\` represents 1	`&` represents 6
`(` represents 2	`%` represents 7
`@` represents 3	`/` represents -1
`?` represents 4	

Marine exo-neurologists are particularly excited by the negative digit, and hope that this new insight into cephalopod neurology will lead to significant advances in their science, which is in its infancy.

As you might expect, the octopus system is base 8 in which each place value is 8 times the value to its right, as in the following examples:

`(@&`	is	$2 \times 8^2 + 3 \times 8 + 6 = 158$
`?/--`	is	$4 \times 8^3 + -1 \times 8^2 + 0 \times 8 + 0 = 1984$
`/(\`	is	$-1 \times 8^2 + 2 \times 8 + 1 = -47$

Your task is to take octopus numbers and represent them as standard base 10 numbers.

Input consists of octopus numbers, one per line. Each number consists of a sequence of between one and eight octopus digits. A single '#' on a line by itself indicates the end of input. This line should not be processed.

Output consists of the corresponding decimal numbers, one per line.

Sample Input

```
(@&
?/--
/(\
?
#
```

Sample Output

```
158
1984
-47
4
```

Problem B

~~Problem I~~

Central Target

30 points

Target is a common newspaper word puzzle in which you are given a grid containing nine letters and must find as many words as possible using those letters. For example, the letters in the following grid can be used to form the words 'LINT', 'TILL', and 'BRILLIANT', among others:

L	A	R
B	**I**	T
N	L	I

Words must be at least four letters in length, and may use each letter in the grid at most once. (Thus they can be no longer than nine letters.) Furthermore, they must contain the grid's central letter, in this case 'I'.

Once the puzzle-maker has chosen the letters to put in the grid, the choice of which is to be the central letter makes a great difference to the number of words that can be made, and hence the difficulty of the puzzle.

You are the puzzle-maker's assistant. You have been given the puzzle-maker's dictionary of valid words, and must analyse potential grids to see how many words from the dictionary can be formed from them. On each grid, the puzzle-maker has not yet decided which letter to put in the central position, and would like to know which choices would permit the smallest and largest numbers of words to be formed.

Given a potential grid, you must find out which letter or letters, when placed in the central position, permit the fewest words from the dictionary to be formed according to the rules. And you must find out just how many words that is. And similarly you must find out which letter or letters placed centrally permit the most dictionary words to be formed.

The input file consists of a dictionary of up to 200,000 words followed by any number of grids to be analysed against that dictionary.

The dictionary consists of words with between four and nine uppercase letters, one per line, given in alphabetical order. A single '`-`' on a line by itself indicates the end of the dictionary.

Subsequent lines each contain one grid to be analysed. Each of these lines consists of exactly nine uppercase letters. A single '`#`' on a line by itself indicates the end of input.

For each input grid, output one line containing the central letter(s) that produce the smallest number of words and that number itself, and the central letter(s) that produce the greatest number of words and that number itself. Where there is more than one letter that produces the minimum or maximum, output all the appropriate letters in alphabetical order (without duplicates, even if a letter appears in the grid more than once—such as I or L in the grid pictured). Separate each number and group of letters with a single space.

[*Turn over for sample data*]

Sample Input

```
APPLE
BANANA
BANE
BRILLIANT
LINT
PALE
PROBLEM
TILL
TRILL
-
LARBITNLI
LEPAPBNNA
LEPAPBNAM
#
```

Sample Output

```
AB 1 ILT 4
BN 1 AE 3
M 0 AE 3
```

Problem C

~~Problem J~~ Adjacent Mastermind 30 points

Mastermind is a game played with a supply of pegs of various colours, or in the absence of proper equipment, pen and paper (or a computer!) using letters `A`, `B`, `C`, etc, as ‘pegs’ with the different letters representing different colours. One player chooses some particular arrangement of coloured pegs or letters and keeps it hidden. The other players attempts to guess the arrangement, guided by a score that the first player determines for each guess.

In ordinary Mastermind, the score is in two parts: a ‘black score’ counting the number of pegs that match the target peg in the same position, and a ‘white score’ that is the number of pegs that are not themselves ‘black’, but match the colour of an otherwise unmatched target peg in a different position from the guess peg.

Adjacent Mastermind adds a ‘grey score’ that is the number of pegs that do not match their corresponding target pegs but can be matched up with otherwise unmatched target pegs in the positions immediately to their left or right. (The leftmost and rightmost guess pegs of course only have one slot that is adjacent to them and that might make them grey.) The white score then becomes the number of pegs that are not themselves ‘black’ or ‘grey’, but match an otherwise unmatched target peg that is at least two positions away from the guess peg.

As in ordinary Mastermind, each target peg may only be matched by at most one guess peg, and each guess peg may only contribute to one of the scores at most once.

For example:

Target	`A B C D`	
Guess 1	`A A A A`	1 black, 0 grey, 0 white
Guess 2	`A C E B`	1 black, 1 grey, 1 white
Guess 3	`E F B B`	0 black, 1 grey, 0 white
Guess 4	`A C B C`	1 black, 2 grey, 0 white
Guess 5	`B E A A`	0 black, 1 grey, 1 white
Guess 6	`A B C D`	4 black, 0 grey, 0 white

In guess 1, only the `A` in slot 1 contributes to the score, since only one peg may match the target `A` and this one is the best match. Similarly in guess 3 only the `B` in slot 3 scores, and similarly only one of the `C`s in guess 4 scores. Finally in guess 5, only one of the `A`s counts as white, because there is only one target `A` available.

Adjacent Mastermind is theoretically easier for the guessing player because more information is provided in response to each guess, but more difficult for the first player because the scoring is more complicated. Your task is to help the first player by calculating each guess’s score.

Input consists of lines containing a target arrangement and a guess arrangement, separated by a single space. Each arrangement is a string of between 2 and 50 uppercase letters, ‘`A`’ to ‘`Z`’. On each line, the guess arrangement will be of the same length as its target.

A single ‘`#`’ on a line by itself indicates the end of input. This line should not be processed.

[Continued overleaf]

Output will be one line for each target/guess input line, containing the guess and its score in the format '*guess*`:` *b* `black,` *g* `grey,` *w* `white`'.

Sample Input

```
ABCD AAAA
ABCD ACEB
ABCD EFBB
ABCD ACBC
ABCD BEAA
ABCD ABCD
ABABAA BCBCAA
#
```

Sample Output

```
AAAA: 1 black, 0 grey, 0 white
ACEB: 1 black, 1 grey, 1 white
EFBB: 0 black, 1 grey, 0 white
ACBC: 1 black, 2 grey, 0 white
BEAA: 0 black, 1 grey, 1 white
ABCD: 4 black, 0 grey, 0 white
BCBCAA: 2 black, 2 grey, 0 white
```

Problem D

~~Problem K~~ **Chemistry 101** 30 points

Chemistry is all about reactions—you throw a bunch of stuff into a test-tube, heat it up, hoping that it will neither explode or poison you, then cool it down and try to work out whether what you have is what you expected. That's the easy (and fun) bit—much harder is recording it all. As is usual with skills of this type, chemistry instructors the world over rely on drill—a seemingly endless set of reactions that the students have to complete. The trick is that everything that appears on the left side (the reagents, or 'input') must appear on the right side (the products, or 'output'). This ought to be simple, but generations of chemistry students have demonstrated otherwise.

Professor Plumbius is getting tired of writing the same comments on his student's worksheets over and over and he wants to automate the process. He wants to be able to enter the equations as written by the students and have the computer produce the comments automatically, thus giving him more time to dream up more equations to give his students to practise on. This is where you come in.

Write a program that will read in a chemistry equation and determine whether it is balanced. If it isn't, your program must tell the student what elements are out of balance and by how much.

Normally a chemistry equation is written like this:

$$2H_2O + SO_2 \longleftrightarrow H_2 + H_2SO_4$$

but due to the limitations of computer input we will present it like this:

```
2H2O + SO2 = H2 + H2SO4
```

This example shows the essentials of an equation: each side consists of one or more molecules, separated by '`+`' signs (the spaces are optional). Each molecule may have a multiplier before it which specifies how many instances of that molecule take part in the reaction. A molecule consists of one or more elements. Each element has a symbol, which is either an uppercase letter, e.g., '`H`', or an uppercase letter followed by a lowercase letter, e.g., '`Br`'. A symbol may be followed by a multiplier specifying how many atoms of that element are present in that part of the molecule. Thus the first term says that there are two instances of a molecule consisting of two atoms of H and one atom of O. (This happens to be water, but you do not need to know that.)

Given that these are exercises handed out to the students, the left hand sides are, by definition, correct. Thus your job is to determine whether all the atoms that appear on the left also appear on the right. If they do, then the equation is balanced. If not, you must report which elements have been created or lost and how much of each.

Input will consist of a number of equations, each on a line by itself. Each line will contain no more than 250 characters. Each equation represents a set of reagents and a set of products, separated by an '`=`' sign. Each set will consist of one or more molecules, possibly with multipliers, separated by '`+`' signs. There may be zero or more spaces on either side of the '`+`' and '`=`' signs.

The last line of input will be a '`#`' on a line by itself. This line should not be processed.

[*Continued overleaf*]

There will be at least one line of output for each equation in the input. If the equation is balanced this line will say '`Equation` *n* `is balanced.`', where *n* is the equation number (starting from 1). If the equation does not balance, then the output line will say '`Equation` *n* `is unbalanced.`' and will be followed by a series of lines of the form

`You have` ⟨`created` *or* `destroyed`⟩ *m* ⟨`atom` *or* `atoms`⟩ `of` *element*.

where *element* is the symbol of the element concerned, *m* is the number of extra or missing atoms of that element, and '`atom(s)`' is singular or plural as appropriate. For each unbalanced equation, these lines should be ordered alphabetically by element symbol and terminated by a blank line. (Note that this means that the final line of your output may be a blank line.)

Sample Input

```
2HBr + H2O+SO2=H2+H2SO3+He
2H2O + SO2 = H2 + H2SO4
2H2O+SO2=H2+H2SO4
#
```

Sample Output

```
Equation 1 is unbalanced.
You have destroyed 2 atoms of Br.
You have created 1 atom of He.

Equation 2 is balanced.
Equation 3 is balanced.
```

Problem E

~~Problem L~~ **Limit Checking** 30 points

From time to time, customers of the First Goldfields Bank of Pembroke make dramatic errors with their banking transactions. For example, they might add an extra zero and transfer $10,000 to someone else when they only intended to transfer $1,000. The bank theorises that this is related to the large sums of money and large drinking budgets associated with a successful goldfield, and plans to introduce transaction limits to detect and prevent serious errors.

Each transaction is a request to transfer an amount of money from one account to another. There are two different kinds of transactions: if the transfer is between two accounts owned by the same customer, then it is an *inter-account transfer* (IAT); otherwise, when transferring money to someone else's account, the transaction is known as a *payment*.

The bank has invited each customer to specify a *maximum instruction limit* and a *daily exposure limit* for each kind of transaction, with the expectation that most customers will want to set higher limits for inter-account transfers than for general payments.

These limits are applied as follows:

- A transaction will fail if its value exceeds the applicable maximum instruction limit.
- A transaction will fail if the applicable daily exposure limit is exceeded when its value is added to the total value of the customer's previously successful transactions of the same kind that day. (But later transactions might succeed if they are for smaller amounts.)

Your task is to write the program to enforce these limits.

Each line of the input file is a *banking record*, consisting of a number of fields separated by commas (`,`). There are four types of banking record, distinguished by the first field:

- '`1`' records are customer records, which have six fields. The second field is the *customer name*, consisting of exactly eight uppercase letters. The remaining fields are amounts; from left to right they are the customer's IAT maximum instruction limit, IAT daily exposure limit, general payment maximum instruction limit, and general payment daily exposure limit.
- '`2`' records are account records, which have three fields. The third field is the *account number*, consisting of exactly six digits ('`0`' to '`9`'). These records specify that the account is owned by the customer named by the second field.
- '`5`' records are instruction records and have a total of six fields, representing a transaction. The second through sixth fields are: the date of the transaction, in the format `YYYYMMDDhhmmss`; the customer making the transaction; the source account (from which the money is to come); the amount; and the destination account.
- The '`9`' record terminates the input file, and has just one field.

[*Continued overleaf*]

All *amounts* in the input data are dollars and cents values written with a decimal point (`.`) and two cents digits (but no commas), with a maximum value of $9,999,999.99. All '1' records appear before all '2' records, which appear before all '5' records, which appear before the '9' record.

All customer names appearing in '2' or '5' records will be valid customers who have been listed in exactly one '1' record, and all account numbers appearing in '5' records will be valid accounts which have been listed in exactly one '2' record. Furthermore, the '5' records will appear in increasing datestamp order. There will be no more than 50 customers and 200 accounts.

The bank does not accept transactions between 23:00 and 06:00, believing that customers will be more than usually impaired during these periods—especially at weekends. Hence such times do not appear in the input file.

Output must contain one line for each input '5' record, starting with '`INSTRUCTION` *n*`: `', where *n* is the instruction number (starting from 1), followed by one of the following messages, as appropriate:

'`NOT OWNER`' if the source account is not owned by the customer;
'`IAT MAX EXCEEDED`' or '`PAYMENT MAX EXCEEDED`' if the transaction amount exceeds the applicable maximum instruction limit;
'`IAT DEL EXCEEDED`' or '`PAYMENT DEL EXCEEDED`' if the transaction amount would cause the applicable daily exposure limit to be exceeded;
or '`IAT OK`' or '`PAYMENT OK`' if the transaction is successful.

If a transaction fails both limit tests, output only the '`...MAX EXCEEDED`' message.

Sample Input

```
1,ROARIMEG,10000.00,20000.00,5000.00,5000.00
1,ASPINALL,1000000.00,5000000.00,250000.00,1000000.00
2,ROARIMEG,123456
2,ROARIMEG,123457
2,ASPINALL,246800
2,ASPINALL,246801
5,20060729101537,ROARIMEG,123456,1000.00,246800
5,20060729111600,ASPINALL,246800,2000000.00,246801
5,20060729151537,ROARIMEG,123457,4500.00,246801
5,20060730084813,ROARIMEG,123457,4500.00,246801
9
```

Sample Output

```
INSTRUCTION 1: PAYMENT OK
INSTRUCTION 2: IAT MAX EXCEEDED
INSTRUCTION 3: PAYMENT DEL EXCEEDED
INSTRUCTION 4: PAYMENT OK
```

Problem F

~~Problem M~~

Flattening Tables

100 points

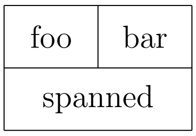
Mensa Web Design Ltd specialises in creating table-based HTML layouts for corporate clients. As a new employee at Mensa, you have been asked to tackle a table simplification problem that has been affecting the company. In short, rather than deal with complicated sets of nested HTML tables, Mensa would prefer that they were flattened into single tables containing equivalently laid-out cells.

For the purposes of this task, you will be working with a subset of HTML even smaller than the subset that Mensa generally uses. Your subset includes only very simple text and well-formed tables, and can be described by the following grammar:

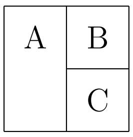
TABLE	→	`<table>` *ROW*... `</table>`
ROW	→	`<tr>` *CELL*... `</tr>`
CELL	→	`<td` *ROWSPAN COLSPAN*`>` *CONTENTS* `</td>`
ROWSPAN	→	empty | `␣rowspan="`*NUMBER*`"`
COLSPAN	→	empty | `␣colspan="`*NUMBER*`"`
CONTENTS	→	*TABLE* | *TEXT*
TEXT	→	a sequence of alphanumeric characters
NUMBER	→	a non-empty sequence of digits with value ≥ 2

('...' indicates that the preceding grammar element may appear in the expansion zero or more times. Whitespace in the grammar is for clarity only; the only whitespace that actually appears in this HTML subset language is the spaces shown explicitly as '␣' in the ROWSPAN and COLSPAN elements.)

All ROWs in a TABLE will each contain the same number of CELLs, except for when cells are omitted due to preceding spanned cells above them or to their left. For example, the following snippets of HTML source code specify the tables shown alongside them. Notice how in the second example the first cell in the second row is omitted and '`<td>C</td>`' actually defines the row's second cell.



```
<table><tr><td>foo</td><td>bar</td></tr>
<tr><td colspan="2">spanned</td></tr></table>
```



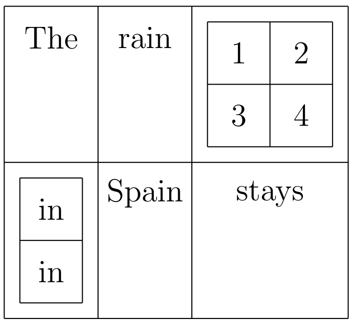
```
<table><tr><td rowspan="2">A</td><td>B</td></tr>
<tr><td>C</td></tr></table>
```

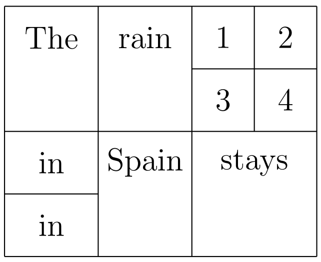
Tables can be nested, which occurs when the CONTENTS for one or more cells is another TABLE rather than simply being TEXT. Mensa's graphic design department produces layouts which never use row or column spans (so their cell start tags are always simply '`<td>`') but often use nested tables. They will even nest tables within several cells in different areas of a table, but never more than one on the same row or column of the enclosing table.

[*Continued overleaf*]

For example, this HTML source code produces the nested layout shown on the left:

```
<table><tr><td>The</td><td>rain</td><td><table><tr><td>1</td>
<td>2</td></tr><tr><td>3</td><td>4</td></tr></table></td></tr>
<tr><td><table><tr><td>in</td></tr><tr><td>in</td></tr></table>
</td><td>Spain</td><td>stays</td></tr></table>
```





The layout on the right is a single 4 by 4 table containing cells that are laid-out equivalently to the nested layout on the left. (The cells are not the same size or shape, but that is something that will be taken care of by the graphic designer in the final tidying up.) This layout is produced by the following HTML source code:

```
<table><tr><td rowspan="2">The</td><td rowspan="2">rain</td>
<td>1</td><td>2</td></tr><tr><td>3</td><td>4</td></tr>
<tr><td>in</td><td rowspan="2">Spain</td>
<td rowspan="2" colspan="2">stays</td></tr>
<tr><td>in</td></tr></table>
```

Your task is to take nested table layouts as produced by the graphic design department and transform them into equivalent layouts using a single table, introducing row and column spans as necessary.

Input will consist of a line containing ‘`<body>`’, followed by any number of lines each containing a nested table layout, one per line, followed by a line containing ‘`</body>`’.

Each line except the first and last will contain no more than 10,000 characters, and will consist of a sequence of ‘`<table>`’, ‘`<tr>`’, ‘`<td>`’, ‘`</td>`’, ‘`</tr>`’, ‘`</table>`’, and alphanumeric tokens matching a valid TABLE according to the grammar above. This table will contain no cells with row or column spans, but may contain nested tables. Tables may be nested up to 10 deep, and the resulting equivalent flattened table will contain at least one row and one column and no more than 100 rows and 100 columns. Each TEXT sequence will contain no more than 100 alphanumeric characters.

Output must consist of a line containing ‘`<body>`’, followed by one line for each input table layout line, followed by a line containing ‘`</body>`’.

[*Continued overleaf*]

Each output line except the first and last must contain a flattened table layout equivalent to the nested layout on the corresponding input line. These lines must contain a valid TABLE according to the grammar above, but with the only allowable CONTENTS being TEXT (i.e., without any nested sub-tables); in particular, all 'HTML tags' must be in lowercase, and row and column spans must have a single space and quotation marks (and must be omitted if their NUMBER would be '1'), as shown in the grammar.

Sample Input

```
<body>
<table><tr><td>A</td><td><table><tr><td>B</td><td>C</td></tr></table>
→ </td></tr></table>
<table><tr><td>The</td><td>rain</td><td><table><tr><td>1</td><td>2</td>
→ </tr><tr><td>3</td><td>4</td></tr></table></td></tr><tr><td><table>
→ <tr><td>in</td></tr><tr><td>in</td></tr></table></td><td>Spain</td>
→ <td>stays</td></tr></table>
</body>
```

Sample Output

```
<body>
<table><tr><td>A</td><td>B</td><td>C</td></tr></table>
<table><tr><td rowspan="2">The</td><td rowspan="2">rain</td><td>1</td>
→ <td>2</td></tr><tr><td>3</td><td>4</td></tr><tr><td>in</td>
→ <td rowspan="2">Spain</td><td rowspan="2" colspan="2">stays</td></tr>
→ <tr><td>in</td></tr></table>
</body>
```

[*The sample input and output shown each contain only four lines. Very long lines have been wrapped for display on the page, as indicated with '→'.*]

Problem G

~~Problem N~~

Big Brother

100 points

A spy agency wants to monitor all communications in a computer network. They have a budget for at most 10 installations of spying software on 10 of the host computers on the network. For the software to work properly each communication line A–B must have at least one host A or B being monitored.

Input will consist of a number of network scenarios. Each scenario will contain:

- An integer n ($10 \leq n \leq 2500$) on a line on its own, giving the number of hosts in the network.
- A line of data for each host (thus n lines in total) giving the list of other hosts to which the host has a direct communication line. The hosts are named $0, 1, \ldots, n-1$; the first line of data gives the neighbours of host 0, the second those of host 1, and so on up to host $n-1$.

  Each of these lines consists of an integer d ($1 \leq d < n$) which is the number of neighbours host k has, followed by d integers which are the neighbouring hosts' names, separated by spaces. The neighbours will be given in numerical order, and will each be valid host names in $0, 1, \ldots n-1$ other than k.

  (Note that each of these input lines may thus be up to 2500×5 characters in length.)

The last line of input will be a '`0`' on a line by itself. This line should not be processed.

Output will consist of one line for each input network, indicating whether the network can be successfully spied upon by infecting 10 nodes. Each line of the output will consist of '`Network` n`: `', where n is the scenario number, starting at 1, followed by '`yes`' or '`no`'.

[*Turn over for sample data*]

Sample Input

```
11
5 1 3 5 8 10
5 0 2 4 6 9
5 1 3 5 6 10
4 0 2 4 6
4 1 3 5 7
5 0 2 4 6 8
6 1 2 3 5 7 9
4 4 6 8 10
4 0 5 7 9
4 1 6 8 10
4 0 2 7 9
11
10 1 2 3 4 5 6 7 8 9 10
10 0 2 3 4 5 6 7 8 9 10
10 0 1 3 4 5 6 7 8 9 10
10 0 1 2 4 5 6 7 8 9 10
10 0 1 2 3 5 6 7 8 9 10
10 0 1 2 3 4 6 7 8 9 10
10 0 1 2 3 4 5 7 8 9 10
10 0 1 2 3 4 5 6 8 9 10
10 0 1 2 3 4 5 6 7 9 10
10 0 1 2 3 4 5 6 7 8 10
10 0 1 2 3 4 5 6 7 8 9
12
11 1 2 3 4 5 6 7 8 9 10 11
11 0 2 3 4 5 6 7 8 9 10 11
11 0 1 3 4 5 6 7 8 9 10 11
11 0 1 2 4 5 6 7 8 9 10 11
11 0 1 2 3 5 6 7 8 9 10 11
11 0 1 2 3 4 6 7 8 9 10 11
11 0 1 2 3 4 5 7 8 9 10 11
11 0 1 2 3 4 5 6 8 9 10 11
11 0 1 2 3 4 5 6 7 9 10 11
11 0 1 2 3 4 5 6 7 8 10 11
11 0 1 2 3 4 5 6 7 8 9 11
11 0 1 2 3 4 5 6 7 8 9 10
0
```

Sample Output

```
Network 1: yes
Network 2: yes
Network 3: no
```

Problem H

~~Problem Q~~ **Half-Score Orienteering** 100 points

Orienteering involves running through unfamiliar territory, using map and compass to navigate to various 'control points' marked on the map. In the most common form of the sport, the order in which the control sites must be visited is set in advance by the race organisers, and the winner of a race is the runner who visits all the controls in the prescribed order and arrives at the finish line in the shortest amount of time. Thus the goal is to visit all controls (in order) as fast as possible.

Another discipline is *Score Orienteering*, in which the goal is the converse: to maximise your score by visiting as many controls as possible in a set amount of time. Here the organisers choose a time limit and assign a points value to each control, generally correlated to its difficulty and distance from the start. A runner's score is the sum of the scores of all the controls visited, less a penalty for arriving at the finish line in excess of the prescribed time limit. (Note that while visiting or passing through a control site more than once is allowed, its score only counts towards the runner's score once, so there's no advantage in revisiting controls.)

Thus runners begin this kind of race by estimating how far they can run in the time available and choosing a subset of (ideally high-scoring) controls that can be arranged into a route with a total length very close to, but not exceeding, their distance estimate.

Finally, an obscure variant of the score discipline is *Half-Score Orienteering*, in which the organisers define the order in which the control sites must be visited, but the runners choose a subset of the controls—aiming for a high-scoring subsequence that forms a route following the general outline of the organisers' course (and ordering) but skipping undesired controls, with a total length close to but not exceeding their distance estimate.

After a race, they are always curious as to whether they could have done better by aiming for a different subsequence of the available controls. Your task is to write a program to determine the maximum score available to each runner in each of a series of races.

Input will consist of a number of race scenarios. Each scenario consists of:

- A line containing an integer, n, the number of controls in the race ($1 \leq n \leq 30$).
- n lines of control data, each containing three integers separated by spaces: 'x y s', where (x, y) are the coordinates in metres of the control site ($-5000 \leq x, y \leq 5000$) and s is the control's score ($10 \leq s \leq 200$).
- One line for each runner in the race, terminated by a line containing only '`# 0`'. Each of these lines contains the runner's name and an integer d, separated by a space. The name is a single word of up to 60 characters, and d is the distance in metres that the runner can travel in the time available ($0 \leq d \leq 10000$).

The last line of input will be a '`0`' on a line by itself. This line should not be processed.

[*Continued overleaf*]

The start and finish points for each race are at the origin $(0, 0)$. Assume that each map area is flat, and that the runners navigate perfectly along straight-line paths between control points (and to/from the start and finish). Then viable routes for a runner start at the origin, pass through the coordinates of some subsequence of the available controls in the order in which they appear in the input, and return to the origin, all with a total length $\leq d$.

(A route longer than d might score enough extra points to outweigh the lateness penalty, but the runners find late finishes so embarrassing that none of them will consider such routes to be viable.)

Output for each scenario will start with a line '`Race` *r*' stating the race number, starting with 1. This will be followed by one line for each of the runners in the race, in the order in which they appear in the race scenario, containing the runner's name and the score of the highest scoring viable route available to that runner, formatted as '*name*: *score*'.

Sample Input

```
5
750 -800 30
1500 0 50
750 750 60
-1250 750 70
-1000 -500 50
Chris 7000
Karl 6500
Tania 5000
# 0
4
500 0 10
0 500 10
-500 0 10
0 -500 10
Hanny 2100
Lizzie 1800
# 0
0
```

Sample Output

```
Race 1
Chris: 230
Karl: 180
Tania: 140
Race 2
Hanny: 20
Lizzie: 20
```