Problem A Simple Graphs 10 points

In the early days of computing the impact printer was often the only readily accessible output device. Graphs were plotted using characters printed in the approximate positions of the desired points. To allow for large graphs, the x-axis would run vertically down the pages while the y-axis ran horizontally across the page. The program would print the axes and scales and the numbers along the axes, while a separate procedure prepared an array holding the relevant characters. You are to write a program to simulate this part of the package.

The input will consist of specifications for several different graphs. The first line for each graph will consist of a pair of integers giving the number of rows and columns necessary to print this graph. You may assume that 1 ≤ rows ≤ 20 and 1 ≤ columns ≤ 20. A line consisting of 0 0 will terminate the input. The specification of each curve in the graph will then follow, with each specification on a single line by itself. Each curve on the graph will be specified as a character (other than '#') followed by several pairs of (x, y) coordinates specifying where that character is to appear. You may assume that 1 ≤ x ≤ rows and 1 ≤ y ≤ columns. Each curve will be terminated by the coordinate pair 0 0. Each graph is terminated by a line containing only the character '#'. There will always be at least one curve in each graph.

The output consists of a series of graphs, one for each specified in the input. Each graph is labelled with its number in the input sequence, followed by the graph in the format described above with the position (1, 1) at the top left of your output. If more than one character is specified for any position, print only the last one specified. Leave one blank line between graphs.

Sample input

```
11 11
\ 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10 11 11 0 0
/ 1 11 2 10 3 9 4 8 5 7 6 6 7 5 8 4 9 3 10 2 11 1 0 0
+ 6 6 0 0
#
5 11
* 2 1 2 2 2 3 2 4 2 5 2 6 3 6 3 7 3 8 3 9 3 10 3 11 0 0
% 5 1 5 2 5 3 5 4 5 5 5 6 5 7 5 8 5 9 5 10 5 11 0 0
#
0 0
```

Sample output

```
Graph 1
\         /
 \       /
  \     /
   \   /
    \ /
     +
    / \
   /   \
  /     \
 /       \
/         \

Graph 2

******
     ******

%%%%%%%%%%%
```

Problem B Compatibility ratings 10 points

Extensive research has shown that the compatibility of two people can be determined from their names. The government wishes to test the compatibility of couples applying for marriage licences (so that those who score less than around 40% can be told they are probably wasting their time). You have been asked to assist by writing a program that computes compatibility scores.

Here's how the system works. We start by counting the number of 'L', 'O', 'V', 'E' and 'S' letters (in either case) in both names. The counts are then concatenated together to form a string representing a 5 (or more) digit number. We then enter an iterative process whereby the next number is formed by adding neighbouring pairs of digits in the current number. The iteration stops when we have a one or two digit number, which is the compatibility percentage, or the process has continued for more than 20 iterations, in which case the compatibility percentage is assumed to be 100%.

For example consider Jenny Shipley and Winston Peters for whom the counts are 1, 1, 0, 4 and 3, which then produce:

```
11043
 2147
 3511
  862
  148
  512
   63
```

So, Jenny Shipley and Winston Peters are 63% compatible.

Write a program to read pairs of names and for each pair calculate the compatibility rating.

Input consists of pairs of lines, with one name (a string of up to 20 letters and spaces) on each line and is terminated by a line containing only a single '#'.

Output consists of the string "Compatibility rating of" followed by a space, the compatibility rating and a '%' sign, for each pair of lines in the input. Follow exactly the output format used in the sample data below.

Sample Input

```
Jenny Shipley
Winston Peters
Helen Clark
Jim Anderton
Bag Sady
Wizards
#
```

Sample output

```
Compatibility rating of 63%
Compatibility rating of 91%
Compatibility rating of 2%
```

Problem C   Palindromes   10 points

A palindrome is a string of characters that reads the same forwards and backwards. A palindrome is stored in run length form as 1 or more (character, run length) pairs. For example: L 1 E 1 V 1 is LEVEL in string form. Notice how the final term (V 1) appears just once in the string form, whereas all of the other terms appear twice.

Write a program to read a series of run-length encoded palindromes, and write them in string form.

Input consists of a number of palindromes, one per line, terminated by a line containing only a single zero (0). Each line starts with an integer N (1 ≤ N ≤ 10) that specifies the number of terms. Each term consists of a character and an integer count (which could be 0). Single spaces are used to separate data items. The file is terminated by a line containing only a zero (0).

Output consists of the string version of each palindrome in the input, on a line by itself. The output string will never contain more than 80 characters.

Sample Input

```
3 L 1 E 1 V 1
4 L 1 E 1 V 1 # 0
2 + 4 - 10
0
```

Sample Output

```
LEVEL
LEVVEL
++++----------++++
```

Problem D Triangles and Lines 10 points

Write a program to determine whether three given points are all the same, lie on a straight line, or form a triangle.

Input consists of a series of cases and is terminated by a line containing 6 zeroes (0 0 0 0 0 0). Each case consists of a line containing 6 integers – three pairs of (x, y) coordinates representing the three points. All coordinates are in the range 1 to 1,000,000.

For each case in the input:
print the word "Point" if all three points are the same,
print the word "Line" if two points are the same or all three points lie on a line,
otherwise print the word "Triangle"

Sample input

```
3 3 4 6 5 9
3 4 3 4 5 5
1 1 1 4 4 4
2 1 2 1 2 1
0 0 0 0 0 0
```

Sample output

```
Line
Line
Triangle
Point
```

Problem F Vote counting 30 points

Voting to select one candidate from two or more is a well known activity. This problem involves counting votes to determine the winner (the candidate with the most votes), and the margin of victory (the difference between the number of votes for the winner and the number of votes for the second ranking candidate). If this difference is zero, then the result is a tie and this should be stated.

Input will consist of details of a number of elections and is terminated by a line containing only ##. Each election consists of a number of votes terminated by a line containing only a #. A vote is merely the candidate's name — a string of up to 20 lowercase letters — on a line by itself. The number of candidates is between 2 and 50 (both numbers inclusive) and there is always at least one vote for each candidate.

Output consists of one line for each election in the input. The line will contain either the name of the winner and the winning margin (the difference between the number of votes cast for the winning candidate and the next ranked candidate) or the names of the candidates involved in a tie, in alphabetical order, separated by 'and'. Follow exactly the output format used in the sample data below and note the use of the singular form when appropriate.

Sample input

```
smith
smith
black
smith
#
jones
tom
tom
green
jones
green
#
jones
tom
tom
jones
green
#
green
green
white
#
##
```

Sample output

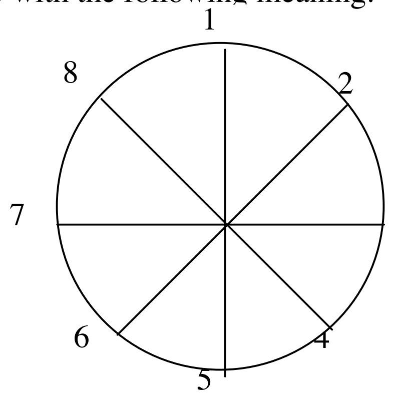
```
smith wins by 2 votes.
a tie between green and jones and tom.
a tie between jones and tom.
green wins by 1 vote.
```

NZ Programming Contest 2000

Problem G Tracks 30 points

A track is the path followed by a ball bouncing around a rectangular container under certain conditions. The ball can travel up, down, left, or right, or along one of the diagonals. The current direction of the ball is given by an integer in the range 1 to 8 with the following meaning:

1	Up
2	Up-right
3	Right
4	Down-right
5	Down
6	Down-left
7	Left
8	Up-left



There are three cases to consider for how a ball bounces off the sides of the container:

1. If the direction of the ball is up, down, left, or right it bounces back in the opposite direction. For example, if it was going up it bounces back down.
2. If the ball is travelling diagonally and it hits a corner then it bounces back in the opposite direction.
3. If the ball is travelling diagonally and it hits a wall then it changes direction by 90 degrees. For example, if a ball traveling up-right hits the right wall of the container its new direction is up-left.

Write a program that draws the track followed by a ball under these conditions, given the size of the container and the initial position and direction of the ball.

Input consists of details of a number of containers and is terminated by a line of five zeroes (0 0 0 0 0). The details of a container are given on a line containing 5 integers. The first two integers give the number of rows and columns that make up the container, in the range 2 to 20 (inclusive). The next two integers give the starting row and column number of the ball, where the coordinates of the top left hand corner are (1, 1), and the fifth integer gives the initial direction of the ball (an integer in the range 1 to 8). Note that it is guaranteed that the starting point lies within the container.

For each container in the input, there should be a display of the container in the output showing the track taken by the ball. Once a ball starts moving it travels endlessly. The output shows all positions occupied by the ball as it moves around its track. Each position contains either a # character (this position is on the track) or a – character (this position is not on the track). There is a blank line after each container display.

Sample Input

```
4 5 3 1 2
3 5 1 5 6
0 0 0 0 0
```

Sample output

```
#-#-#
-#-#-
#-#-#
-#-#-

#---#
-#-#-
--#--
```

Problem H Soccer League 30 points

Write a program that, given the results of a series of soccer games, determines the final league table at the end of the competition. The results for each game are given using the following format:

```
    team1 g1 team2 g2
```

where *team1* and *team2* are the names of the two teams involved, *g1* is the number of goals scored by *team1* and *g2* is the number of goals scored by *team2*. If *g1* = *g2* then the result is a draw, otherwise the team that scores the most goals wins. Note that a team cannot play itself.

During a competition each team plays a number of games. For each win they receive 3 points, and for each draw they receive 1 point. At the end of the competition, the teams are ranked by:

1 The number of points (the more the better).
2 For teams with the same number of points, the number of wins (the more the better).
3 For teams equal on tests 1 and 2, the number of goals scored by the team minus the number of goals scored against the team (the bigger the better).
4 For teams equal on tests 1, 2 and 3, the number of goals scored (the more the better).
5 For teams equal on tests 1, 2, 3 and 4, list the teams in alphabetical order of team names.

Your program is to output each league table, sorted according to the above rules, from first to last.

Input consists of a number of sets of results and is terminated by a line containing only ##. Each set of results consists of one or more lines in the format given above and is terminated by a line containing only #. All team names consist of up to 20 lower case letters and there will never be more than 20 teams. The maximum number of games occurs when playing a Round Robin, i.e. when each team plays all the other teams once. The minimum number of goals scored by a team in a game is 0 and the maximum is 99.

Output consists of a league table for each set of results in the input. Each league table consists of a series of lines specifying the teams, in the order specified above. Each line contains the name of the team, left justified in a field of width 20, followed by the number of games won, the number of games drawn, the number of goals scored by the team, the number of goals scored against the team, and the number of points earned by the team, all right justified in fields of width 5. A blank line appears after each league table.

Sample Input

```
foo 1 blarg 2
blarg 0 gonzo 0
gonzo 1 foo 3
#
##
```

Sample Output

```
blarg                   1    1    2    1    4
foo                     1    0    4    3    3
gonzo                   0    1    1    3    1
```

Problem I Vampire Numbers 30 points

An n-digit number is a number which requires exactly n decimal digits to write it, that is $10^{n-1} <= x < 10^n$. A (2n)-digit number V is a vampire if it is the product of two numbers $0 < P <= Q < 10^n$ such that the decimal digits of V are a permutation of the decimal digits of both P and Q (assume that P and Q are first padded on the left with enough zeros to make them exactly n digits). For example, 12310938 is a vampire number because 12310938 = 1338 * 9201.

Write a program that will read in a series of numbers and, for each of them, determine the next Vampire number, i.e. the smallest vampire number that is greater than it.

Input will consist of a sequence of integers, in the range 1..99999999, one per line. The sequence will be terminated by a line containing a single zero (0).

Output will consist of a series of numbers, one for each number in the input. In each case it will be the smallest Vampire number larger than the given number. If there is no 8 digit Vampire number larger than the one you are given, output "No Vampires".

Sample Input

```
12310937
99999998
0
```

Sample Output

```
12310938
No Vampires
```

Problem K Machine Emulator 100 points

Your employer has unearthed some assembly language programs written for an ancient processor, the DODO-II, that has long since become extinct. The company needs some way of running these programs, and you have been asked to write an emulator for the instruction set.

DODO-II Assembly programs had a very simple format. Each line consists of one of the following:

- nothing, or only spaces and tabs,
- a comment (a string of text starting with a % sign, possibly preceded by spaces and/or tabs),
- a instruction possibly followed by a comment (as above),
- a label possibly followed by a comment (as above), but no instruction.

Comments and blank lines are ignored. An instruction consists of an op-code (defining the operation to be performed) followed by zero or more operands separated from each other by commas and from the op-code by one or more tabs or spaces. A label consists of a string of between 1 and 20 upper case letters, starting in column 1 and terminated by a ':'.

The DODO-II had 16 registers, named R1 to R16, each of which held a signed 32 bit integer stored in two's complement format. Three arithmetic operations were supported: addition, subtraction and multiplication. The syntax for these three instructions is:

```
    ADD   DestReg, RegOrVal1, RegOrVal2
    SUB   DestReg, RegOrVal1, RegOrVal2
    MUL   DestReg, RegOrVal1, RegOrVal2
```

Each RegOrVal was either a register name or an integer constant. The two RegOrVal's were combined (for subtraction RegOrVal2 was subtracted from RegOrVal1) and the result stored in register DestReg.

A value can be loaded into a register or moved from a register to another using the instruction:

```
    MOV   DestReg, RegOrVal1
```

Several branch instructions are available. All specify the name of a label. The 6 conditional branch instructions evaluate a condition and execution then continues at the branch address (label) if the condition is true, or at the next instruction if the condition is false. BRU always branches to the label.

```
    BEQ   label, RegOrVal1, RegOrVal2   % RegOrVal1 = RegOrVal2
    BNE   label, RegOrVal1, RegOrVal2   % RegOrVal1 ≠ RegOrVal2
    BLT   label, RegOrVal1, RegOrVal2   % RegOrVal1 < RegOrVal2
    BLE   label, RegOrVal1, RegOrVal2   % RegOrVal1 ≤ RegOrVal2
    BGT   label, RegOrVal1, RegOrVal2   % RegOrVal1 > RegOrVal2
    BGE   label, RegOrVal1, RegOrVal2   % RegOrVal1 ≥ RegOrVal2
    BRU   label                         % branch unconditionally
```

The INP and OUT instructions perform I/O and the HLT instruction terminates execution.

```
    INP   DestReg                       %DestReg ¨ next input value
    OUT   SrcReg                        %Outputs value in SrcReg
    HLT                                 %halts the program
```

Input consists of a number of (program, input) pairs. Each program is syntactically valid, and is terminated by a line containing only a #. You may assume that none of the arithmetic operations result in an overflow condition.

The lines following the program contain the input data for the program in the form of one integer per line. Each time an INP statement is executed an integer is read from the input. The end of the input is marked by a line consisting of '#'. It is guaranteed that sufficient input data is provided. Any unused input should be skipped over. The end of input is signified by an empty program, i.e. one that has '#' as its first line.

Output consists of the values generated by any OUT instructions executed by the program. A blank line follows the output produced by each program.

Sample input

```
% Compute x^y

    INP R1      % Read x
    INP R2      % Read y
    MOV R3, 1
LOOPTOP:
    BLE END, R2, 0%A legal but ugly comment
    MUL R3, R3, R1
    SUB R2, R2, 1
    BRU LOOPTOP
END:
    OUT R3
    HLT
#
-2
5
#
#
```

Sample output

```
-32
```

Problem L NZ Voting 100 points

The current New Zealand voting system is a proportional system known as Mixed Member Proportional (MMP). Each voter has two votes: a party vote (for the party which they want to see form the government) and an electorate vote (for the person they wish to see representing them in Parliament). The proportions of the overall party vote determines the number of seats each party has in Parliament, whereas in each electorate the candidate with the most electorate votes is elected as a member of Parliament (MP). Electorate MPs that stood as party candidates form the basis of that party's Parliamentary representation. Additional seats are allocated as necessary to provide parties with their entitlement from the party vote.

The total number of seats that each party is entitled to is determined using the Saint-Lague formula. Only parties that receive 5% or more of the party vote OR that win at least one electorate seat are considered — all other parties are ignored. If any seats are won by independents or parties that were not listed on the official list of parties offered to voters, then those seats are deducted from the number available to be shared out. If the number of available seats is *N*, we form the *N* odd numbers 1, 3, 5, … $2N-1$ and calculate *N* quotients for each party by dividing their list votes by these numbers. The highest *N* of all these quotients are listed and each party gets the same number of seats as it has quotients in this list.

For example, assume that the White party has 1023 party votes, the Red party 777 votes, the Black party 600 and the No-hopers 12 (the No-hopers did not win an electorate seat either) and that there are 11 seats available. In this situation, the 11 divisors are the numbers 1, 3, 5, 7, ..., 21 and these produce the following quotients:

```
White: 1023.0, 341.0, 204.6, 146.1, 113.7, 93.0, …, 48.7
Red:    777.0, 259.0, 155.4, 111.0,  86.3, 70.6, …, 37.0
Black:  600.0, 200.0, 120.0,  85.7,  66.7, 54.5, …, 28.6
```

The highest 11 quotients are:

1023.0(W), 777.0(R), 600.0(B), 341.0(W), 259.0(R), 204.6(W), 200.0(B), 155.4(R), 146.1(W), 120.0(B), 113.7(W),

and from this we can see that White gets 5 seats, Red 3 and Black 3. If two quotients are equal, then the quotient emanating from the party with the most party votes is deemed to be bigger. You may assume that no two parties ever get the same number of votes.

One final complication is that a party may win more electorate seats than the total number it qualifies for based on its party vote. This is known as an overhang. For example, assume there were 7 electorates in our example (leaving 4 list seats), of which White won 2 and Red 5, so that Red has won more electorate seats than the number of seats it is entitled to under the Saint-Lague formula. Red gets to keep the 5 electorate seats it has won. White gets the 2 electorate seats it has won plus 3 list MPs. Black gets 3 list MPs. The size of parliament is expanded by 2 (the number of 'extra' seats won by Red) to a total of 13. This expansion lasts until the next election.

Write a program to read in details of a number of elections and to determine the number of seats that each party has won. The input is supplied in three parts:

1 *N*, the total number of seats in Parliament ($3 \le N \le 1000$ — a value of 0 signifies the end of input).

2 the number of electorate seats won by each party that won electorate seats. Each of these lines consists of a party name (a character string of up to 20 uppercase letters that contains no spaces) followed by an integer. If the total number of electorate seats is S, then $2 \le S \le N$. A line consisting of a single '#' terminates this section.

3 the number of party votes received by each party. If a party is not listed in this section it means that it did not offer itself as a list party. The format is as for the number of electorate seats. Note that the total number of party votes across all parties will be less than 2,000,000,000.

Output consists of the results for each election represented in the input. Each election result consists of the number of seats won by each party represented in Parliament followed by a blank line. The list is in alphabetical order, with the party name followed by a space followed by the number of seats won.

Sample Input

```
20
NATS 5
LABS 4
UNTIED 1
JADES 2
#
LABS 50100
NATS 30200
UNTIED 50
JADES 11250
NZLAST 8400
NOTQUITE 1000
#
0
```

Sample Output

```
JADES 2
LABS 10
NATS 6
NZLAST 2
UNTIED 1
```

Problem M Duckworth-Lewis 100 points

The game of cricket is widely played in many Commonwealth countries. The one-day version goes something like this. Team 1 bats for 50 overs and tries to score as many runs as possible. Team 2 then bats for 50 overs and tries to score more runs than team 1. If both teams score the same number of runs then there is a tie, otherwise the team with the most runs wins.

Each team has eleven batsmen. When a batsman is 'out' the batting team loses a 'wicket', and the batsman who is out is replaced by another. When the batting team loses 10 wickets during its 'innings' it is 'all out' and its innings finishes immediately, even if some of the 50 overs remain.

If time is lost due to rain, and the number of overs available to one or both teams is reduced, then it becomes necessary to adjust (up or down) the number of runs that must be scored by team 2, using the concept of resource availability. If a batting side bats for the full 50 overs, then it has had 100% of its resource (its ability to score runs) available to it. If a side bats for fewer than 50 overs, then it has had less than 100% of its resource available to it.

The Duckworth-Lewis Method of resource allocation is based on a table that maps combinations of overs remaining and wickets lost to resource percentages. Consider the following extract from the table:

Wickets lost		0	2	5	7	9
Overs left	50	100.0%	83.8%	49.5%	26.5%	7.6%
	40	90.3%	77.6%	48.3%	26.4%	7.6%
	30	77.1%	68.2%	45.7%	26.2%	7.6%
	10	34.1%	2.5%	27.5%	20.6%	7.5%

At any point in an innings, the resource percentage remaining to a batting team is determined by the number of overs left in the innings and the number of wickets lost by the batting team.

Each innings consists of one or more periods spent batting. Consider the following period. At the start of the period 40 overs remained and the team had lost 0 wickets. After 10 overs rain halted play for a while. During the 10 overs, 2 wickets were lost. Using the table, we see that the resource percentage at the start of the period was 90.3%, and the resource percentage at the end of the period was 68.2%. Consequently, the batting team used 22.1% of its resources during the 10 over batting period.

Now we can describe how to calculate the target for team 2. Let *ru1* be team 1's resource usage and *ru2* be team 2's resource usage (the amount used so far plus the amount available from team 2's remaining overs, if any). If $ru1 > ru2$, then the number of runs scored by team 1 is multiplied by $ru2 / ru1$, and the result is truncated to an integer. If, on the other hand, $ru2 > ru1$, then we add $(ru2 - ru1)*225$ (truncated to an integer) to the number of runs scored by team 1. In either case this gives the number of runs necessary to tie — team 2 needs one more run than this to win.

Once the revised target for team 2 has been computed, then there are two possibilities:
1. The game is over, and the result (win for team 1, win for team 2, or tie) can be declared. The game is over if there are 0 overs remaining in team 2's innings, or if team 2 has already scored more runs than their target.
2. The game continues, with team 2 trying to exceed their target in the overs that remain.

Write a program that uses the Duckworth-Lewis Method to determine the outcome of a number of matches. In all cases the innings of team 1 is complete.

The first part of the input file is the table of resource utilisations for various combinations of overs remaining and wickets remaining. Each line in this section starts with the number of overs remaining, which is followed by 10 resource utilisations (expressed as percentages, with a single digit after the decimal point) for 0 wickets lost, 1 wicket lost, and so on up to 9 wickets lost. The table is terminated by a line containing a single 0. While the table may not be complete (in other words, it may not contain data for all overs from 1 to 50) it will contain all the entries needed in the remainder of the problem.

Following the table are a number of match situations that require analysis. Each match situation is described by 5 lines of input:

1. The name of team 1, up to 20 characters long. A line consisting of # terminates the input.
2. The name of team 2, up to 20 characters long.
3. The details of the innings of team 1. The line starts with an integer that specifies the number of batting periods (it will be in the range 1 to 10). Details of the batting periods follow. Four integers specify the details of each period: the number of overs thought to remain at the start of the period, the number of overs actually delivered in the period, runs scored in the period and wickets lost in the period.
4. The details of the innings of team 2. It has the same format as the second line. Note that it is legal for there to be 0 periods for team 2.
5. A single integer specifying the number of overs remaining in team 2's innings between 0 and 50 inclusive.

For each match situation, report either the result (if a result has occurred) or what team 2 must do to win the game. Use the output formats shown in the sample output below exactly.

Notes:

1. Use "over", "run" and "wicket" after the number 1; "overs", "runs" and "wickets" after anything else.

2. In the case of a tie, the name of team 1 is the first listed.

Sample Input

```
50 100.0 92.0 83.8 72.0 62.0 49.5 37.0 26.5 16.0  7.6
40  90.3 83.0 77.6 68.0 59.0 48.3 36.0 26.4 16.0  7.6
30  77.1 73.0 68.2 60.0 52.0 45.7 35.0 26.2 15.0  7.6
10  34.1 33.0 32.5 31.0 29.0 27.5 23.0 20.6 14.0  7.5
5   18.4 18.2 17.9 17.4 16.8 16.4 15.0 14.0 13.0  7.0
0
New Zealand
England
1 50 50 250 5
1 50 40 199 5
0
New Zealand
England
1 50 50 250 5
1 50 40 181 5
0
New Zealand
England
1 50 40 120 5
1 40 30 140 9
5
New Zealand
England
1 50 40 190 7
0
40
India
Pakistan
2 50 10 90 2 30 10 100 3
0
20
#
```

Sample Output

```
England defeats New Zealand by 18 runs.

New Zealand ties with England.

To defeat New Zealand, England needs to score 19 runs,
from 5 overs with 1 wicket in hand.

To defeat New Zealand, England needs to score 215 runs,
from 40 overs with 10 wickets in hand.

To defeat India, Pakistan needs to score 209 runs,
from 20 overs with 10 wickets in hand.
```

Problem N Sorting Mäori 100 points

The order used in Williams' Mäori dictionary is not actually stated, however the following procedure appears to reproduce that order.

1 [Left-to-right] At this stage nothing is known.
Look only at the letters, ignoring alphabetic case and vowel length (macrons). The tricky thing here is that 'ng' and 'wh' are to be treated as single letters, where 'n' < 'ng' < 'o' and 'w' < 'wh' < 'x'. For example, 'wini' < 'whatu'. If a difference is found, stop.

2 [Right-to-left] The letters are known to be the same except for case and vowel length.
Look only at the vowels, and put long vowels before short vowels. So 'käkä' < 'kakä' < 'käka' < 'kaka'. If a difference is found, stop.

3 [Left-to-right] At this stage only alphabetic case and non-letters differ.
Sort into lexicographic order based on the ASCII representation of the characters.

The input to the program is a sequence of lines, each containing a single word, or a phrase consisting of two or more words separated by spaces, without initial or trailing spaces. In order to remain within the ASCII character set, we adopt the common convention, used in most English dictionaries, of indicating long vowels by putting colons after them. Thus we will distinguish between 'ka:ka:', 'kaka:', 'ka:ka', and 'kaka'; and we will write 'Mäori' as 'Ma:ori'. You should treat 'a:', 'e:', 'i:', 'o:', 'u:', 'A:', 'E:', 'I:', 'O:', 'U:' as single letters just like 'ng', 'wh', 'NG', 'WH', 'Ng', 'Wh', 'nG' and 'wH'; the colon is not otherwise allowed in the input. The end of the input is marked by a line consisting of '#'.

The output of the program is the same collection of lines (using the same conventions), but sorted into Williams' dictionary order.

Sample Input

```
wini
kapu
kaka
kA:ka
kahu
whare whui
whatu
matuku-moana
O'Regan
whare nui
ka:ka:
kaka:
n#^%@$#uwa
nuwa
#
```

Sample Output

```
kahu
ka:ka:
kaka:
kA:ka
kaka
kapu
matuku-moana
n#^%@$#uwa
nuwa
O'Regan
wini
whare nui
whare whui
whatu
```